Principles and implementation of a generic synchronization interface between SystemC AMS models of computation for the virtual prototyping of multi-disciplinary systems

Liliana Lilibeth Andrade Porras

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Liliana Lilibeth ANDRADE PORRAS

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse

Principes et réalisation d'une interface de synchronisation interopérable entre modèles de calcul SystemC AMS pour le prototypage virtuel optimisé de systèmes multi-disciplines

soutenue le 12 janvier 2016 devant le jury composé de :

M. Ian O'CONNOR	Rapporteur	École Centrale de Lyon, France.
M. Christoph GRIMM	Rapporteur	TU Kaiserslautern, Allemagne.
Mme. Noëlle LEWIS	Examinatrice	Université de Bordeaux 1, France.
M. Daniel SAIAS	Examinateur	ASYGN, France.
Mme. Alix MUNIER	Examinatrice	Université Pierre et Marie Curie, France.
M. François PÊCHEUX	Directeur	Université Pierre et Marie Curie, France.
Mme. Marie-Minerve LOUËRAT	Co-directrice	Université Pierre et Marie Curie, France.
M. Alain VACHOUX	Invité	École Polytechnique Fédérale de Lausanne, Suisse.

Résumé

La conception de systèmes embarqués devient de plus en plus complexe. Ces systèmes sont hétérogènes dans le sens où ils nécessitent l'intégration de composants décrits au moyen de plusieurs disciplines scientifiques, par exemple, l'électricité, l'optique, la thermique, la mécanique, la chimie ou la biologie. De plus, ces disciplines peuvent être représentées dans des domaines temporels différents, par exemple, le domaine des événements discrets, celui du temps discret, ou celui du temps continu. Face à cette situation, les concepteurs ont besoin d'outils de modélisation et de simulation efficaces pour décrire le comportement d'un système hétérogène dans un environnement de simulation unique.

Nous examinons la possibilité de modéliser, de simuler et de synchroniser les systèmes multi-disciplines dans le même environnement, en utilisant comme référence la norme de simulation « SystemC Analog/Mixed-Signal (AMS) ». Nous analysons la méthode introduite par SystemC AMS pour synchroniser le domaine des événements discrets avec celui du temps discret, et nous identifions ses inconvénients. Nous proposons une formalisation du problème de synchronisation qui permet de détecter les problèmes existants dans un modèle avant la simulation.

Nous proposons un prototype de simulateur appelé « SystemC Multi-Disciplinary Virtual Prototyping (MDVP) », qui est implémenté comme une extension de SystemC. Il permet la modélisation, l'élaboration, et la simulation hiérarchique de systèmes multi-disciplines au moyen de plusieurs modèles de calcul. Pour concevoir le simulateur MDVP, nous introduisons un nouveau principe de synchronisation entre plusieurs modèles de calcul.

En outre, nous introduisons une méthodologie pour ajouter, dans le prototype de simulateur, des modèles de calcul représentés par plusieurs domaines temporels. Nous appliquons cette méthodologie pour ajouter un modèle de calcul « Timed Data Flow (TDF) » dans SystemC MDVP. Ce modèle de calcul repose sur la sémantique du temps discret introduite par SystemC AMS, et sur la formalisation du principe de synchronisation entre le domaine des événements discrets et celui du temps discret.

Nous mettons en œuvre le modèle de calcul TDF, dans le cas d'un capteur de vibrations et son circuit numérique. Ce modèle comporte une boucle d'asservissement et plusieurs interactions entre le domaine des événements discrets et celui du temps discret.

Abstract

The design of embedded systems is currently an increasingly complex problem. These systems tend to become heterogeneous in the sense that they require the integration of components described by means of different physical/engineering disciplines, for example, electrical, optical, thermal, mechanical, chemical, or biological. Besides, these disciplines can be described under different time domains, for example, Discrete Event (DE), Discrete Time (DT), or Continuous Time (CT). To address this problem, designers require modeling and simulation tools to describe the system's components under different time domains and synchronize them in the same simulation environment.

We explore the possibilities of modeling, simulating and synchronizing multi-disciplinary systems in the same environment, using as reference the SystemC Analog/Mixed-Signal (AMS) simulation standard. We analyze the method introduced in SystemC AMS for synchronizing the DE and DT domains, and we identify its drawbacks. Besides, we introduce a new formalization of the synchronization problem, which is used to detect issues in a model before simulation.

We propose a simulator prototype called SystemC Multi-Disciplinary Virtual Prototyping (MDVP), which is implemented as an extension of SystemC. It allows the modeling, and the generic hierarchical elaboration and simulation of multi-disciplinary systems, by means of different Models of Computation (MoCs). To build the MDVP simulator, we introduce a synchronization principle to handle interactions between MoCs.

In addition, we introduce a methodology to add, in the simulator prototype, MoCs described under different time domains. We apply this methodology to add a Timed Data Flow MoC in SystemC MDVP. This MoC implements the DT semantics introduced by the SystemC AMS standard, and is based on the synchronization principle between the DE and DT domains.

Using the TDF MoC, we implement and simulate a case study of a vibration sensor model and its digital front end circuit. This case study includes a feedback loop and several interactions between the DE and DT domains.

Acknowledgements

This thesis represents not only the result of three years of my work, but also the effort of several people, who I wish to thank. I would like to express my deepest gratitude to my advisor, Dr. François Pêcheux, professor at LIP6 laboratory, for giving me the opportunity to work in this interesting research subject, and also for his guidance and support. I would also like to thank my co-advisor, Dr. Marie-Minerve Louërat, CNRS research director and chief of the SoC department at LIP6 laboratory, for her valuable and constructive suggestions during the writing of this manuscript. I am deeply grateful for her time and support all along these last three years.

I would like to thank Dr. Torsten Mähne for his supervision, his valuable help and advices. I am grateful to him for the long discussions that helped me to achieve this thesis work. I would like to extend my gratitude to Dr. Alain Vachoux for his suggestions and the interest shown by my research work.

I would like to thank the members of my thesis committee. I am thankful to Prof. Ian O'Connor and Prof. Christoph Grimm, for reading and evaluating my research work. I am also thankful to Prof. Noëlle Lewis, Mr. Daniel Saias, Prof. Alix Munier and Prof. Alain Vachoux, for their questions, comments and suggestions.

I would like to address special thanks to Dr. Alain Greiner, professor at LIP6 laboratory, and to Dr. Gerard Paez Monzón, professor at the University of the Andes (Venezuela) for opening me the door of the LIP6 and for giving me the opportunity to come in France.

I would like to thank my colleagues at LIP6 laboratory, Benoît V., Vanessa T., Zhi W., Quentin M., Clément D., Hao L., Laurent L., Alexandre B., and Jean-Baptiste B., for their support and the excellent working atmosphere. I am particularly grateful to Cédric Ben Aoun, for his friendship and the daily discussions about work and life.

Finally, I would like to express my deep and sincere gratitude to my parents and sisters, for their continuous love, guidance, support and encouragement. I would also like to thank my boyfriend, César, for his love, care, help and patience during all these years.

Contents

List of Figures

List of Tables

CHAPTER

1 Introduction

Contents

1.1. Context

Nowadays, a large percentage of devices, in addition to having microprocessors embedded and connected with the outside world through sensors and actuators, are connected with other devices thanks to the internet. This means that both physical and virtual worlds are merging [1]. When we talk of internet, we are referring to the evolving entity, growing in importance, which began as *Internet of Computers*, by offering a global network with services as the World Wide Web; which became *Internet of People*, by connecting millions of people through the social networks; and which is becoming in **Internet of Things (IoT)**, by creating an ecosystem where billions of devices are interconnected and communicate with each other [2]–[4]. We are specifically referring to the "technological revolution in the future of computing and communication that is based on the concept of anytime, any place connectivity for anything" [5].

The valuable contribution of the IoT, of merging the physical world and the virtual world, has been possible thanks to the basis provided by the **Cyber-Physical Systems (CPS)**, which are a particular variation of embedded systems including sensors and actuators. Using sensors, the embedded systems process the information from the physical world and make it available for the virtual world; and using actuators, the virtual world can directly impact the physical world [6].

Today, we consider that the *modeling and simulation* of CPS is an increasingly complex problem because they tend to become **heterogeneous**, in the sense that they include components associated to different physical/engineering **disciplines**: electrical, optical, thermal, mechanical, chemical, or biological are some examples. Besides, these disciplines can be described under different timed or untimed **domains**: continuous time, discrete time, synchronous data flow, or discrete event are some examples. This indicates that the challenge in the development of CPS is to bridge the gap among the different included disciplines.

On the one hand, the **modeling** attempts to represent real systems through a set of interconnected elements, which can be described at different abstraction levels, and which can have specific characteristics. On the other hand, the **simulation** ensures that these elements are always executed while respecting temporal semantics, provides mechanisms for sharing data at specific times, and proposes techniques for preserving the integrity of the transmitted information.

We also consider that it is very important that these systems of heterogeneous nature can be properly represented in **virtual prototypes**; which are fast and fully functional executable software models of hardware systems [7]. They can be used in a set of simulation tests, which allow to verify design concepts, and improve the real systems development process.

The components of these functional models require two types of synchronization: (1) **time synchronization**, referring to the information exchanges at the right time, because components can operate at different computation speeds; and (2) **data synchronization**, referring to the information exchanges in the right format, because data transfers can require approximation of values among domains. Moreover, these virtual prototypes can be described and verified using two different approaches: *co-simulation* and *unified multi-domain simulation*.

Co-simulation is the method by which several components or subsystems, described under different design languages or implemented by means of different design tools, are connected together

and simulated in a distributed manner. Although co-simulation can address the interaction between timed or untimed domains, it requires a frequent synchronization between the parallel executions of different simulation environments, affecting significantly the overall simulation performance [8]. In general, interfaces for enabling the co-simulation should be defined. An example of a co-simulation environment, which proposes the interaction between the discrete event and the continuous time domains is presented in [9].

For its part, the **unified simulation** is the approach proposing the joint design and simulation of hardware and software components in the same environment. It should reduce the modeling time, the number of design cycles, the development cost, and the unexpected effects produced by the interactions between components. An example of a unified simulation environment, which addresses the joint design of HW/SW/Analog systems is presented in [10].

Usually, the unified multi-domain simulation approach is expected to be able to define and verify the behavior of systems, whose components are described by means of different timed or untimed domains, without needing to worry about how these components will be synchronized. The idea of making available a simulation environment including these features, is maybe the ideal dream of many designers of heterogeneous systems.

1.2. Objectives and Research Contributions

At present, as several authors have discussed, the modeling and simulation of embedded systems is not an easy challenge: it is difficult to bring, handle and control worlds of different natures together [11]; the mix of analog and digital parts makes the design process more complicated [8]; and the heterogeneity is the major obstacle for developing model-based design tools for these systems [12].

The purpose of this thesis is to explore the possibilities of simulating and synchronizing multi-disciplinary systems with respect to the Discrete Event (DE) time domain, using as reference the simulation standard called SystemC Analog/Mixed-Signal (AMS) [13]. We consider the DE time domain for defining the simulation bases because the representation, processing, transmission and storage of the embedded systems' information is performed by general purpose systems described in the digital world (easily represented by discrete event time behaviors). Another purpose of this work is to make of the heterogeneous simulation a generic process, offering the possibility of coupling and integrating multiple physical/engineering disciplines. The specific contributions of this work are summarized below.

- **Synchronization with the Discrete Event (DE) Domain through the Discrete Time (DT) Domain**: we analyze the only synchronization method included in SystemC AMS and we identify its drawbacks. Thanks to this analysis, we formalize this synchronization method and improve it in two aspects:
  - The detection of synchronization issues during a simulation period is now performed before the simulation phase.
  - Suggestions to solve these synchronization issues are also identified and notified to the designer before the simulation phase.

Moreover, we highlight that the existing synchronization method is not sufficient to support the interactions of several domains with respect to the DE domain.

- **Unified Simulator Prototype**: we propose a new simulator prototype called SystemC Multi-Disciplinary Virtual Prototyping (MDVP), which includes generic methods to prepare and simulate heterogeneous models. The simulator kernel is proposed and implemented as an extension of the system design language called SystemC [14].

- **Addition of Models of Computation (MoCs)**: we introduce a methodology to add to the SystemC MDVP simulator prototype, MoCs described by means of different time domains. In this context, a **Model of Computation** is the term used to define the time abstraction, the computation rules, the semantics of communication and synchronization between processes in a process network [15].

- **Timed Data Flow (TDF) MoC**: we design and implement a simplified version of the TDF MoC described in the SystemC AMS standard [13]. This implementation allows us to validate the methodology proposed to add MoCs in the unified simulator prototype. Moreover, thanks to the DT nature of the TDF MoC, we implement the synchronization method previously formalized between the DE and DT domains.

- **Case Study**: we implement and simulate the case study of a vibration sensor model and its digital front end circuit, using the TDF MoC included in the SystemC MDVP simulator prototype. The model includes a DE feedback loop and multiple synchronization points with respect to the DE domain.

1.3. Thesis Organization

After defining, in Chapter 1, the context and the contributions of our work, this document is organized as described in the following.

In Chapter 2, we present our motivation focused on the modeling and simulation of multi-disciplinary systems. We introduce SystemC, the modeling language based on a Discrete Event simulation kernel; and also the Analog/Mixed-Signal extensions of this language, which allow to simulate discrete time and continuous time behaviors. Additionally, we describe and analyze the SystemC-AMS proof-of-concept simulator [16] for finally defining the problems to be addressed in this thesis.

In Chapter 3, we summarize the state of the art associated with our research. We present several approaches for modeling and simulation of multi-disciplinary systems, we identify the level at which the heterogeneity can be expressed in each approach, if they are able to include different time domains, how such domains are included, and the synchronization methods defined for ensuring their interactions.

In Chapter 4, we explain the issues that can arise during the synchronization interactions between the Discrete Event and Discrete Time domains. We present a formalization of these interactions using a

Coloured Petri Net (CPN) representation [17] and then, we propose a DE-TDF pre-simulation analysis useful to detect the synchronization issues, and offer possible solutions for these issues.

In Chapter 5, we describe the new simulator prototype called SystemC MDVP. We introduce the hierarchical synchronization principle adopted for the representation of interactions, and the generic methods proposed for the elaboration and simulation of multi-disciplinary models. Moreover, in this chapter we include an overview about the implementation of this prototype. Finally, we describe the methodology proposed to add Models of Computation in the SystemC MDVP simulator prototype.

In Chapter 6, we present a simplified version of TDF MoC, which works respecting a discrete time semantics. This MoC not only integrates the synchronization method formalized in Chapter 4, but also validates the methodology proposed in Chapter 5 for adding new MoCs in SystemC MDVP.

In Chapter 7, we show the case study of a vibration sensor model and its digital front end circuit, this model is described using the TDF MoC. In the case study, several TDF blocks contain non-unitary attributes and are interconnected in a TDF cluster, which includes a feedback loop and several interactions with the discrete event domain. In this chapter, we present the DE-TDF pre-simulation analysis applied in the model to detect the synchronization issues and also to propose solutions for such issues.

Finally, in Chapter 8, we conclude this research and give an outlook of the future works.

CHAPTER

2 Motivation and Problem Definition

Contents

2.1. Introduction

The modeling and simulation of heterogeneous systems is becoming an important aspect in the design flow of **Systems-on-Chip (SoC)**, which are integrated circuits including, in a single chip, components associated to different physical/engineering disciplines and described under different time domains. They can integrate and mix, as shown in Figure 2.1, digital parts (processors, memories, interconnection busses, or timers), Radio Frequency (RF) parts (communication or transmission channels), Analog/Mixed-Signal (AMS) parts (converters or sensors), and also physical or mechanical parts.

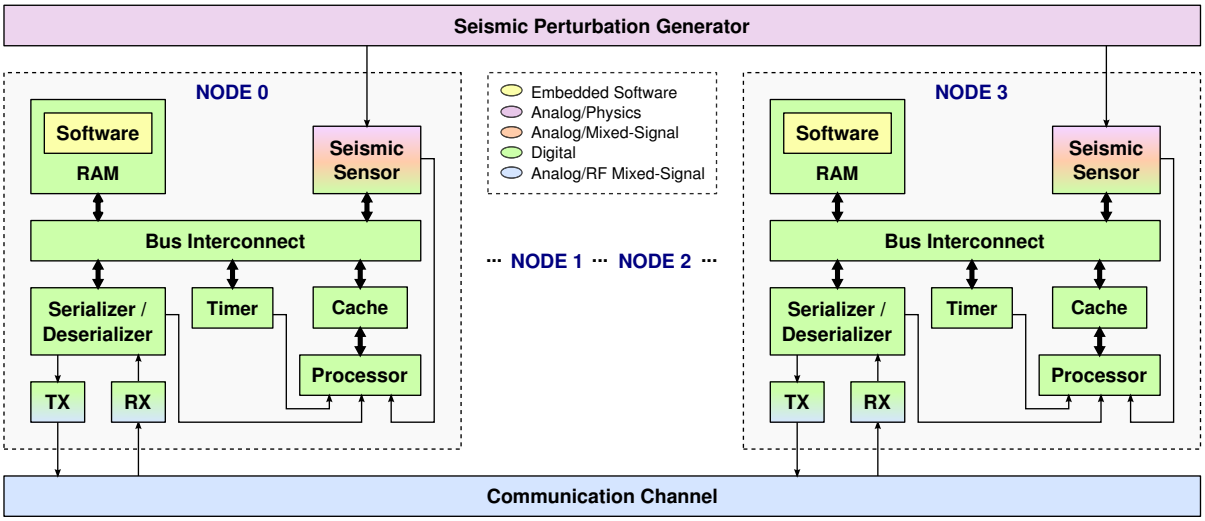

Figure 2.1: Wireless Sensor Network (WSN) application for the determination of the epicentre of a planar seismic perturbation (adapted from [18]).

As the complexity of these systems is increasing, due to the heterogeneity of its components, its design flow requires a parallel and concurrent development of hardware and software, synthesis and verification. This means that design aspects, such as functionality, timing, physical design, and verification should be simultaneously addressed [19].

The need to design these heterogeneous components in a same environment is increasing: the independent modeling and simulation of the embedded parts involves the use of dedicated tools. These tools allow the isolated verification of components, and involve a complex and very costly design process. Interactions among parts should be analyzed, and carefully integrated to avoid an impact in the embedded system behavior.

The AMS extensions for the design and modeling languge called SystemC, were proposed to address this need. They facilitate the understanding of the complexity of heterogeneous embedded systems before its expensive fabrication. Using these extensions, models and applications can be described at different abstraction levels, and can be implemented using different time domains.

At present, the abstractions of time, computation, communication and synchronization offered by the AMS extensions of SystemC, are not sufficient for representing the behavior of complex multi-disciplinary systems. The main drawback is that mechanisms to incorporate new abstractions to these extensions are missing. Additionally, the problem of detecting synchronization issues caused by the interactions among domains, has not been carefully analyzed.

In order to understand the specific problems to be addressed in this thesis, in Section 2.2, we present an introduction to the SystemC modeling language, its main features, and the elaboration and simulation semantics that make SystemC an extensible language. In Section 2.3, we describe the generalities of the SystemC AMS extensions, its architecture, models of computation, solvers, and synchronization methods. Moreover, we introduce the SystemC-AMS Fraunhofer Proof-of-Concept (PoC) simulator currently implemented, we identify its drawbacks, and we analyze how the main Model of Computation (MoC) included (Timed Data Flow (TDF) MoC) works in this simulator. In Section 2.4, we present the problem statement, and finally in Section 2.5, we conclude the chapter providing an overview of how the problems will be addressed.

2.2. SystemC

SystemC [20]–[22] is a system design modeling language, which adds to C++ a library created to address the modeling of both hardware and software systems. On the one hand, it is considered a *system level specification language*, which allows the modeling at the algorithmic level. On the other hand, it is considered a *hardware description language*, since it allows modeling of systems above the Register-Transfer Level (RTL) of abstraction.

The advantage of SystemC over other hardware description languages refers to the different abstraction levels offered: in the same language a system can be described in a high abstraction level, and can be progressively refined. Other languages do not support the modeling of high abstraction levels, e.g. Transaction Level Modeling (TLM) [23]. Another advantage is the verification environment offered where C and C++ code can be easily integrated.

SystemC includes important hardware oriented features: (1) a global discrete **time model**, represented by 64 bits of resolution and whose progress is handled by a *simulation kernel*; (2) a **concurrency concept**, which refers to the concurrently execution of multiple processes supported by a cooperative multitasking model (*scheduler*); (3) **hardware data types**, supporting explicit bit widths for integer and fixed point quantities; (4) a **hardware hierarchy** implemented by constructs (*modules*); and (5) a **communication and synchronization** model implemented by different mechanisms (*interfaces*, *ports* and *channels*). These features are supported by the language architecture presented in Figure 2.2.

In this section we introduce the concepts required for understanding this thesis work. We focus on the description of the SystemC core language elements, the SystemC Discrete Event (DE) simulation kernel, and its operation phases. Thanks to these phases, the modeling language could be extended for supporting behaviors described in other time domains than DE.

2.2.1. Core Language Elements

SystemC follows a block-oriented approach in the sense that it allows the representation of systems by a combination and interconnection of blocks and signals: *blocks* represent particular or complex behaviors, and they can have multiple inputs and outputs; and *signals* ensures the communication among blocks. We consider that this approach is very interesting because users, without long experience in the design of electronic systems, can easily represent and simulate particular behaviors.

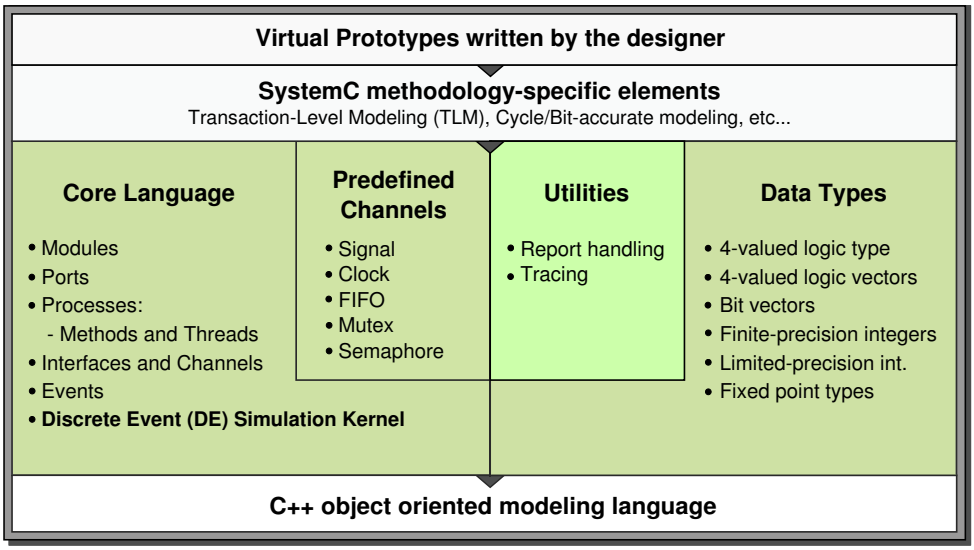

Figure 2.2: SystemC Language Architecture (adapted from [14]).

In SystemC, as shown in Figure 2.3, the primitives which allow designers to partition models, and break complex systems into simpler sub-systems, are called **modules**. They may contain *ports*, *interfaces* and *channels*; and they can be hierarchical, this means that they may contain *processes* and other modules instantiated within them.

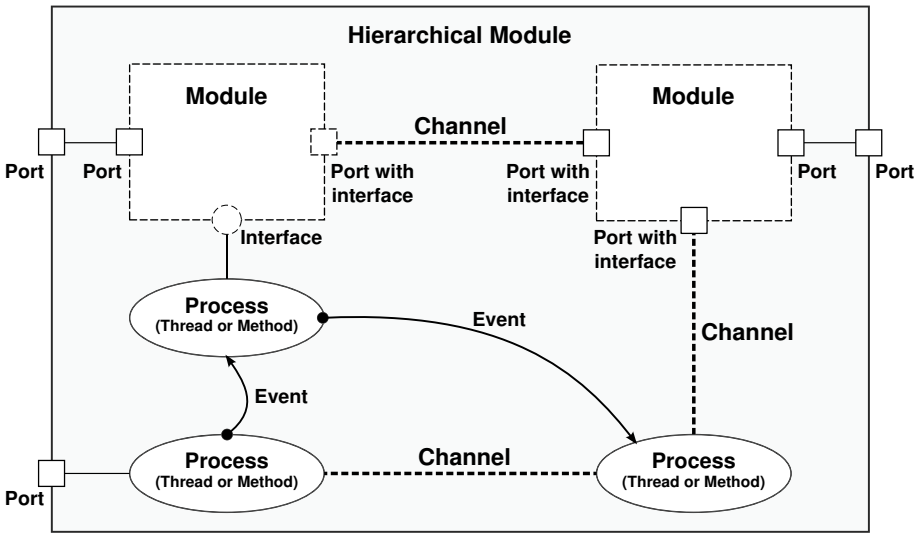

Figure 2.3: SystemC Components (adapted from [20]).

On the one hand, **ports** are the objects through which the module communicates with other modules and its environment. They are responsible for calling methods defined outside of modules, in particular defined by *interfaces*. These **interfaces** define sets of methods to access the **channels**, which are containers (e.g. *FIFOs* or *signals*) maintaining the modules' state and allowing communication, they hold and transmit data. Channels are responsible for implementing methods defined by *interfaces*. In brief, *ports* are connected to *channels* through *interfaces*.

On the other hand, **processes** describe the operation of the modules, and provide mechanisms for simulating concurrent behaviors. They are specific functions implemented by the designer and called

from the *DE kernel* during simulation. Two kind of processes can be defined in SystemC: (1) **methods**, which are always executed from beginning to end; and (2) **threads**, which can suspend itself during simulation using *wait* statements. These kinds of processes are also known as **static processes** because they are registered in the *DE kernel* before simulation. There is also the possibility of creating processes during simulation, in this case they are called **dynamic processes** [24]. The mechanism for creating dynamic processes will be presented in Chapter 6.

Processes can communicate using channels (e.g. *signals*), or using **events**, which are the objects able to determine whether and when a process should be triggered or resumed. The control of *events* is handled by the *DE simulation kernel*.

2.2.2. Discrete Event (DE) Simulation Kernel

The **DE simulation kernel** of SystemC provides the core features for the *elaboration* and *simulation* of models [25]. **Elaboration** creates the data structures required to support the simulation semantics: creates the module hierarchy, instantiates processes, bounds ports and channels, and sets the time resolution to be used (by default is 1ps). **Simulation**, runs the *scheduler* and deletes the data structures created during elaboration. The elaboration and simulation semantics defined by the SystemC standard are summarized in Figure 2.4.

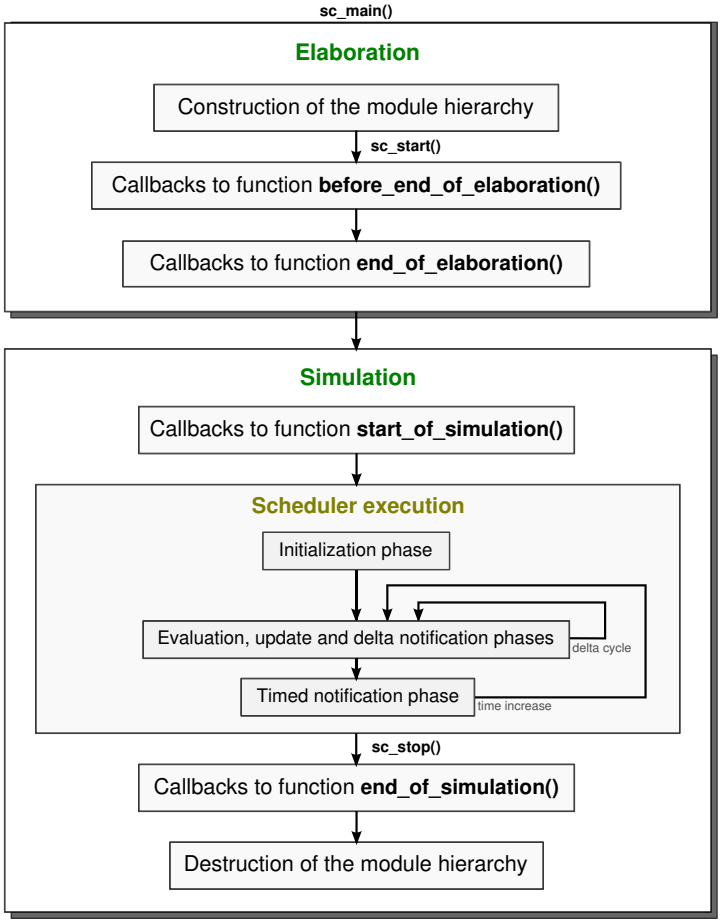

Figure 2.4: SystemC Elaboration and Simulation Semantics.

The **scheduler** is the heart of SystemC, it controls the timing and the order for executing the processes. The scheduler execution is performed in five phases: (1) *initialization*, where all defined processes are entirely executed (*methods*) or until the first *wait* statement (*threads*); (2) *evaluation*, where each process ready to run is selected and its execution is resumed (this may cause new processes ready to run in the same phase); (3) *update*, where channels are updated thanks to the results of the evaluation phase; (4) *delta notification*, where are analyzed the notifications made during the previous two phases: if they should be executed in the current simulation time then, the evaluation phase is re-executed; (5) *timed notification*, where the notifications are also evaluated: if they should not be executed in the current simulation time, such time is increased and then, the evaluation phase is re-executed. When no more notifications are present, the scheduler execution is stopped.

An important feature of the scheduler is that it supports the notion of **delta cycle**, which is an infinitesimal increase of time used to impose a partial order of the simulation actions. When the scheduler processes a *delta cycle*, it executes actions that are scheduled at the current time in the three consecutive *evaluation*, *update* and *delta notification* phases. At a particular simulation time, multiple *delta cycles* may occur.

In addition to the phases to create/destroy the module hierarchy, and to perform the scheduler execution, the SystemC standard offers four callbacks or virtual functions that can be overloaded by objects in the module hierarchy (*modules*, *ports* and *channels*) for allowing the applications to perform further elaboration and simulation actions. These callbacks are introduced below.

- before_end_of_elaboration(): it allows to perform elaboration actions depending on the properties of the module hierarchy, which can still be modified in this stage. The instantiation of modules, ports and channels; the port binding; and the instantiation of static processes are also allowed. Using this callback, for example, some modules could be instantiated to monitor the module hierarchy.
- end_of_elaboration(): it allows to perform elaboration actions, which do not need to modify the module hierarchy. In this stage, the instantiation of objects derived from the SystemC modules, ports, and channels; and the creation of static and dynamic processes are allowed. Using this callback, for example, an application can perform rule checking, diagnostics about the module hierarchy, and internal actions to prepare the hierarchy for simulation.
- start_of_simulation(): it allows to perform actions at the start of simulation, for example: to open stimulus and files, or to print additional diagnostic messages. In this phase the instantiation of objects derived from the SystemC modules, ports, and channels; and the creation of dynamic processes are allowed.
- end_of_simulation(): it allows to perform actions at the end of simulation, for example: to close files and to print simulation results. In this phase SystemC objects cannot be instantiated, and new processes cannot be created.

The four callbacks previously introduced are very important because they make SystemC an extensible language. We will take advantage of this fact for making the heterogeneous simulation a generic process.

2.3. SystemC Analog/Mixed-Signal (AMS) Extensions

The **AMS extensions of SystemC** were created for increasing the capabilities of the modeling language to allow the design, simulation and verification of not only digital software and hardware systems, but also of analog/continuous time behaviors. Therefore, they attempt to address the needs from the telecommunication, automotive, and semiconductor industry [26].

These extensions are defined as a C++ standardized library, which follows the same block-oriented approach of SystemC to allow the creation of multi-disciplinary models, that can be simulated in the Discrete Event (DE), Discrete Time (DT), and Continuous Time (CT) domains. They were standardized by the Accellera Systems Initiative organization [27] with the specific purpose of providing: a methodology for modeling embedded AMS systems [28], and also a complete definition of the AMS class library so that a SystemC AMS implementation can be developed [13]. At present, only one implementation of these extensions is available, it is the SystemC-AMS Proof-of-Concept (PoC) library [16] developed by the Fraunhofer Institute for Integrated Circuits IIS [29].

2.3.1. SystemC AMS Language Standard Architecture

Due to the heterogeneity involved in the complex embedded systems designed today, different description styles and Models of Computation (MoCs) should be combined within a system. Therefore, the architecture of the SystemC AMS language standard, as shown in Figure 2.5, is structured following a layered approach [30].

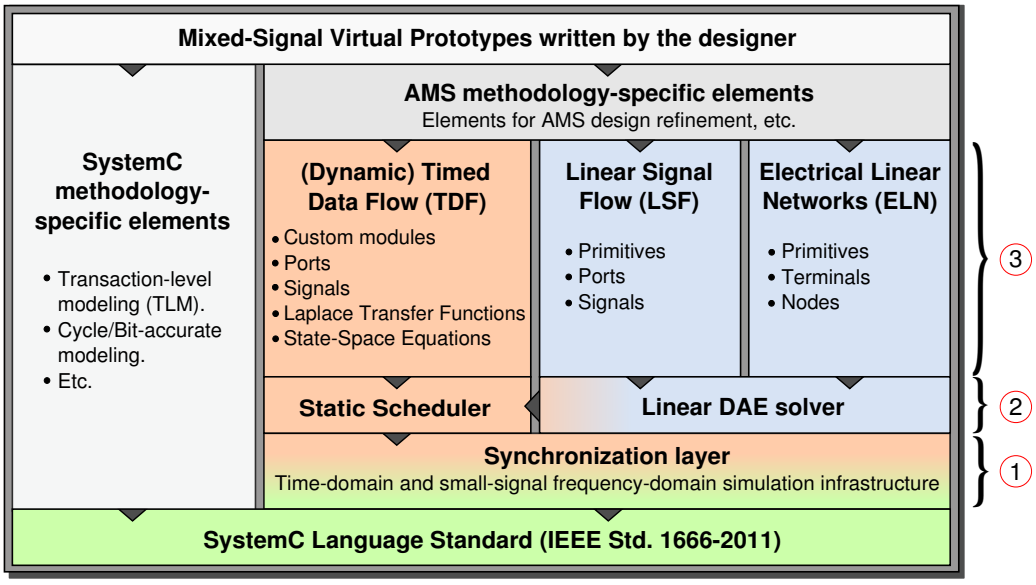

Figure 2.5: SystemC AMS Language Standard Architecture (adapted from [28]).

First, the *synchronization layer*, indicated with (1) and constructed on top of the SystemC standard, is responsible for scheduling the SystemC AMS simulation: it determines the time points at which the digital and analog simulations are synchronized, it activates each solver, and it performs the communication among the different solvers. A **solver** in SystemC AMS, is the object instantiated not only for computing the solution of systems by mathematical methods, but also for performing the specific elaboration and simulation phases associated to a MoC.

Second, the *solvers layer*, indicated with (2) and constructed on top of the synchronization layer, computes the behavior of analog blocks and contains the algorithms proposed for solving specific systems. Third, the *view layer*, indicated with (3), provides the interfaces used by the designer to write executable models, e.g. procedural behaviors or netlists. Besides, it contains the methods (accessible by solvers) for defining the structures to be used by each MoC during simulation.

MoCs integrated in SystemC AMS attempt to follow the layered architecture, and represent the set of rules for defining the behavior and interactions among AMS components. The Timed Data Flow (TDF) MoC, allows discrete time modeling, and efficient simulation of signal processing algorithms and communication systems at the functional and architectural level; the Linear Signal Flow (LSF) MoC, supports modeling of continuous time behaviors through a set of predefined primitives for non-conservative system descriptions; and the Electrical Linear Network (ELN) MoC enables modeling of electrical networks, also in the continuous time domain.

Despite the three MoC independent formalisms and the well-separated layered architecture proposed by the standard, some drawbacks are present in the SystemC-AMS PoC implementation during the synchronization with the DE domain, and the addition of new MoCs. These aspects are discussed below.

a. SystemC-AMS PoC Synchronization

Based on the principle of describing continuous time behaviors to be embedded in timed data flow clusters (set of interconnected timed data flow modules), SystemC-AMS allows the communication and synchronization with the DE domain only through the TDF MoC.

The current implementation of the synchronization layer includes a Synchronous Data Flow (SDF) algorithm, which uses a static scheduler for determining the order in which the AMS modules should be executed, and the order in which the analog solvers should be activated during simulation.

Although this implementation can be efficiently simulated at high abstraction levels [31], it imposes restrictions for the other MoCs included in the prototype. Only one synchronization mechanism (by means of TDF MoC) is available between the DE kernel and the existing MoCs, then, all MoCs are always executed under the control of the TDF MoC, which imposes temporal semantics for synchronization. This means that the time resolution in other MoCs is limited by the time resolution of the TDF MoC.

b. SystemC-AMS PoC Extension

At present, the MoCs included in the PoC simulator are not sufficient for representing the behaviors of complex multi-disciplinary systems: extensions require new formalisms for describing, for example, electromechanical or fluidic behaviors.

The drawback in SystemC AMS is that mechanism to add new MoCs is not well defined. Only programmers and experienced designers, with an extensive knowledge of the current implementation, can include new solvers and synchronization methods [32].

Although there is not any document explaining the mechanism required to add new MoCs, two extensions have been proposed. The first [33], introduces a MoC enabling the modeling of non-linear networks; and states that in networks where DE, DT, and CT models are coupled, the synchronization becomes more complex. In this case specific details about the synchronization implementation are not provided. The second [34], introduces a MoC facilitating the unified description of the power transfer within parts of heterogeneous systems, thanks to the Bond Graph formalism. The addition of this MoC is only based on the PoC simulator's internal details, which are not clearly specified in the standard. Due to the importance of the TDF MoC during the synchronization and the addition of MoCs in SystemC-AMS, the TDF MoC should be carefully analyzed.

2.3.2. Timed Data Flow (TDF) Model of Computation (MoC) in SystemC-AMS

The **TDF MoC** is based on the SDF formalism [35]. It is described as a DT modeling style that considers data as signal, which values are sampled with constant time steps. It was created with the aim of offering an efficient simulation approach for high abstraction levels. TDF not only keeps two important properties of the SDF formalism: the abilities to determine a static schedule, and to perform a periodic execution; but also adds temporal semantics to this SDF formalism, with the purpose of linking it with other timed MoCs. TDF is considered as a powerful modeling style for the creation of AMS descriptions in virtual prototypes, because it processes modules at DT points without directly using the dynamic schedule of the SystemC DE simulation kernel [10].

A TDF model, as shown in Figure 2.6, is basically composed of a set of TDF modules (indicated with (1)) and it can be interconnected using TDF signals (indicated with (2)). Connections among TDF modules and TDF signals are established through TDF ports (indicated with (3)). Sometimes, a TDF model can interact with SystemC (SC) modules (indicated with (4)). In this case, the SC modules which have SC ports (indicated with (5)) are interconnected with the TDF modules using SC signals (indicated with (6)). Connections from SC modules to TDF modules are established through TDF input converter ports (indicated with (7)), and connections from TDF modules to SC modules are established through TDF output converter ports (indicated with (8)). The set of interconnected TDF modules (indicated with (9)) is called TDF cluster. In Figure 2.6, two clusters are presented: the first is composed by **A** and **B** modules, and the second is composed by **C** and **D** modules.

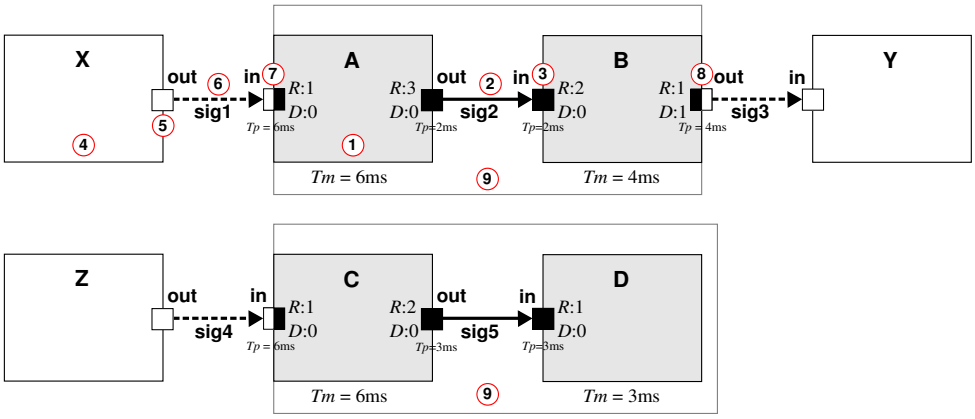

Figure 2.6: Example of a Basic Multirate TDF Model with 2 TDF Clusters, 4 TDF Modules and 2 TDF Signals. It Interacts with the DE Domain Using TDF Converter Ports.

On the one hand, each TDF module is described with one attribute and one processing() function. The attribute is the **module time step** *Tm*, which represents the time period in which the processing() function associated to the same module should be executed. The processing() function, is a mathematical function, which can depend on the module inputs or internal states. At each time step, a TDF module reads a fixed number of samples from each of its input ports, executes the processing() function, and writes a fixed number of samples to each of its output ports.

On the other hand, each TDF port is described with three *attributes*. The first attribute is the **port time step** *Tp*, which represents the time period in which the samples are read or written by a TDF port. The second attribute called **rate** *R*, represents the number of read or written samples by a TDF port during a module time step. The third attribute called **delay** *D*, corresponds to the initial available samples in a TDF port when simulation starts.

Additionally, TDF modules can define two functions: set_attributes(), useful for fixing the TDF module and TDF port attributes previously described; and initialize(), for fixing initial sample values before starting simulation.

Following the SystemC approach, the execution of AMS applications, including TDF modules, is performed in two phases: *TDF elaboration*, executed in the context of a SystemC end_of_elaboration() callback; and *TDF simulation*, registered in the context of a SystemC start_of_simulation() callback, and executed in the first *delta cycle* of the SystemC scheduler. Actions performed during these phases are summarized in Figure 2.7.

During the **TDF Elaboration**, the *TDF attribute settings stage* executes in no particular order all the set_attributes() functions defined by TDF modules. The *TDF time step calculation and propagation stage* computes and propagates a time step value for each TDF port and each TDF module instantiated accordingly to the Equation 2.1, where *Tm* is the time step associated to a TDF module, *Tp* is the time step associated to a TDF port (belonging to preceding TDF module), and *R* is the rate associated the same TDF port. The time step associated to a port determines the time period in which the TDF samples are consumed/produced from/to each input/output TDF port. To achieve this stage at least one time step should be assigned in a module or a port of each TDF cluster included in the application.

$$Tm = Tp * R \tag{2.1}$$

The *TDF computability check stage* determines whether each TDF cluster is computable. First, TDF ports (i and j), bounded by the same TDF signal should respect the Equation 2.2, where q_M is the number of times that the module (to which the TDF port belongs) is activated during a execution period, and *R* is the rate associated to the TDF port. Second, there should exist an activation order (static schedule) that guarantees that each TDF module will be activated the number of times q_M previously determined by a SDF analysis.

$$q_{M_i} * R_i = q_{M_j} * R_j \tag{2.2}$$

The drawback identified during the TDF elaboration is that the DE/TDF interactions are not considered for determining the static schedule of each TDF cluster included in the application. This

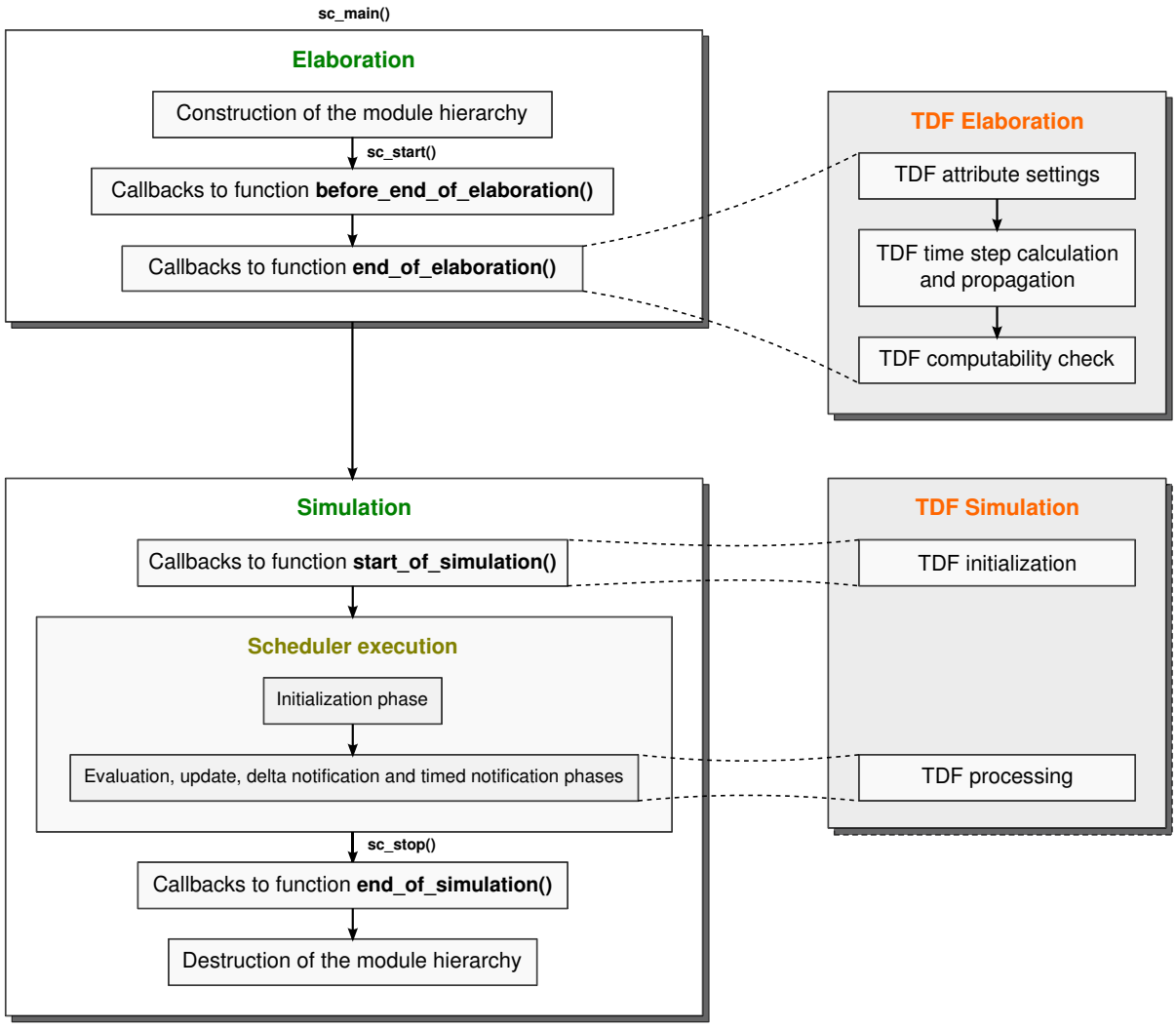

Figure 2.7: SystemC-AMS Elaboration and Simulation Semantics.

means that when the SDF analysis is applied, the read and write operations, performed by the input and output converter ports, are not included in the schedule. The last fact could cause synchronization problems later in simulation.

During the **TDF simulation**, the *TDF initialization stage* executes in no particular order all the initialize() functions defined by TDF modules; and the *TDF processing stage* executes the modules' processing() functions following the TDF schedule determined during elaboration. Unfortunately, as previously mentioned, temporal inconsistencies between the DE and DT domains can be detected in this stage due to the non-inclusion of DE/TDF interactions in the schedule.

The drawback for understanding the particular DT simulation, described by means of the TDF MoC, their interactions with the DE domain, and the detection of synchronization problems, is that the temporal TDF semantics is not formalized. Currently, we do not have precise information about how the time is handled in the TDF MoC, how the schedule is determined, and why all synchronization problems cannot be detected during elaboration.

2.4. Problem Statement

Having in mind the modeling, simulation and verification of muti-disciplinary systems, on the one hand we would like to have a simulation kernel built on top of the SystemC language standard allowing the independent addition of models of computation, that can be associated to different physical/engineering disciplines. In this sense, independent term refers that the simulation kernel is not modified when a new model of computation is added. *Despite that the AMS extensions of SystemC allow the addition of new models of computation, this task is only in the hands of experts: a deep knowledge is needed about how the AMS simulator works, how the synchronization is defined, and how the models should be prepared in each MoC before the simulation phase. Today, we do not have a well-defined method to add any model of computation.* Our idea is to propose a new simulation kernel defining the way in which the elaboration and simulation phases are called on a model, regardless of the different models of computations there involved then, to establish a method for implementing new models of computation, always preserving the same simulation kernel.

At present, the addition of new models of computation is also limited because only one synchronization method is available for synchronizing models of different natures with the discrete event domain, this is the synchronization method between the Timed Data Flow MoC and the SystemC DE simulation kernel. Besides, the addition of new synchronization methods is based on the TDF semantics. This means that when a new MoC is added it should respect the TDF semantics and provide the means to communicate and synchronize through it. To solve this issue, we want to propose an infrastructure to add new synchronization methods that are not forced to always respect the discrete time temporal semantics previously defined by the TDF MoC. The proposition rests on the idea of expanding the current synchronization possibilities.

On the other hand, we want to support the current synchronization method to manage the interactions between the TDF and DE MoCs, and we are interested in improving it. *Actually, the verification and detection of synchronization errors between the TDF and DE MoCs is performed only during the simulation phase, when each module's processing() function is called. This means, that eventually during long-running simulations, the designer must wait long before discovering that his model is wrong.* We believe that one synchronization analysis can be applied during the elaboration phase of models because all the TDF cluster attributes are known, and the accurate synchronization times can be determined in advance before simulation.

A formalization of the synchronization method implemented to synchronize the discrete time and discrete event domains could help to understand how the interactions are performed and when they occur, also it could help to detect the temporal inconsistencies during the execution of a model. *Unfortunately, attempts to formalize this synchronization method has not been carried out until now.*

2.5. Conclusion and Outlook

After introducing in this chapter the SystemC language standard and its AMS extensions, we identify the four issues to be addressed during this thesis: the addition of models of computation in the current simulation prototype is not obvious, the interactions with the discrete event domain can be

performed only through one synchronization method, the detection of synchronization errors in the available synchronization method is performed during the simulation phase instead of the elaboration phase, and there is not an available formalization to correctly analyze the synchronization interactions between the DE and TDF MoCs.

In the next chapters, after analyzing different techniques adopted for the simulation of multi-disciplinary systems (Chapter 3), we follow a bottom-up approach to solve the identified issues. First, we demonstrate that the interactions between the DE kernel and the TDF MoC can be formalized, and then that this formalization can be used to detect and solve the synchronization problems before performing the simulation phase (Chapter 4). Second, we propose a new simulation kernel integrating generic phases for the elaboration and simulation of models, which can involve different timed or untimed domains. In addition, we introduce a mechanism to add new models of computation, where each one has the possibility of integrate multiple synchronization methods (Chapter 5). Finally, as a first attempt to validate the proposed solutions, we present the implementation of a simplified version of the TDF MoC included in the AMS extensions of SystemC (Chapter 6), and also a case study of a model described in the DE and TDF MoCs (Chapter 7).

CHAPTER

3 State of the Art

Contents

3.1. Introduction

In order to identify the features and requirements to be considered for modeling, simulating and synchronizing multi-disciplinary systems, described under different timed or untimed domains, in this chapter, we introduce the state of the art of several simulation approaches based on metamodels, high-level programming languages, and the hardware description language called SystemC. In the next sections, we analyze these approaches by means of three key aspects:

- **Modeling**: we identify the basic elements used for representing models, how these models and their elements can be interconnected to each other, whether hierarchical modeling is allowed, and whether the notions of *computation* and *communication* among the model elements are well-separated. In this sense, **computation** refers to the means provided for encapsulating the information processing; and **communication**, refers to the means provided for transmitting the processed information.
- **Heterogeneity**: we identify the heterogeneity level (*shallow* or *deep*) supported by each framework [36]. These heterogeneity levels are represented in Figure 3.1.

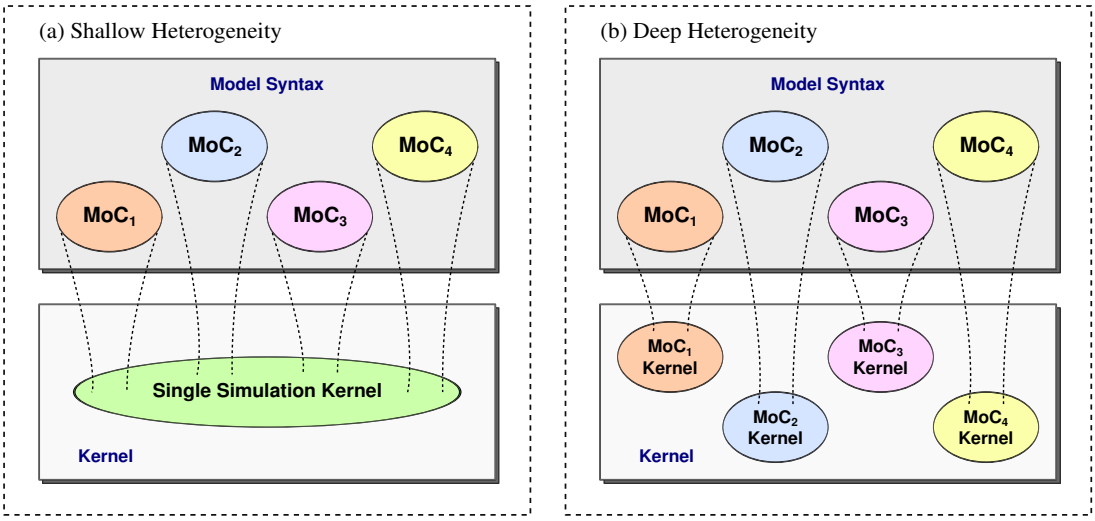

Figure 3.1: Shallow vs. Deep Heterogeneity (adapted from [36]).

  - **Shallow heterogeneity**, is when syntactic extensions providing support for different Models of Computation (MoCs) are only implemented at the *language-level*. It means that there are constructs (types, channels, signals, etc.) in the design language that describe a model following a desired MoC, and that are mapped into a **single simulation kernel**.
  - **Deep heterogeneity**, is when some syntactic extensions are implemented at the *kernel-level*. It means that there are constructs in the design language that describe a model following a desired MoC, and that are mapped into **MoC-specific kernels** responsible for simulating the different components of a model, according to the involved domains.

- **Simulation**: we present the execution semantics required for performing the model execution on each framework.

In Section 3.2, we discuss three simulation frameworks developed by the Center for Hybrid and Embedded Software Systems (CHESS) at University of California, Berkeley [37]: *Metropolis* and *Metro II*, based on metamodels with formal semantics supporting simulation and formal analysis of complex electronic-system designs; and *Ptolemy II*, based on a high-level programming language, and which is considered as the promoter of the heterogeneous hierarchical system design.

In Section 3.3, we present three simulation frameworks (*HetSC*, *HetMoC*, and *ForSyDe*), which address the issue of the concurrent execution of processes, belonging to different MoCs, by means of SystemC-based components: processes, interfaces, and channels.

In Section 3.4, we discuss two frameworks extending the SystemC Discrete Event (DE) simulation kernel: *SystemC-H*, which provides a simulation kernel supporting heterogeneity by means of different models of computation; and *SystemC-A*, which provides a simulation kernel supporting digital and analog behaviors.

Finally, in Section 3.5, we conclude this chapter summarizing the features and requirements that will be considered for defining the means to ensure the multi-disciplinary synchronization, and the bases for a unified and extensible modeling and simulation environment.

3.2. Frameworks Based on Metamodels and High-Level Programming Languages

3.2.1. Metropolis

Metropolis [38], [39] is a platform-based design environment characterized by a flexible and formal semantics which supports simulation and formal analysis of embedded software. It is a specification based on Java, which allows communication between models working at different abstraction levels, and models concurrently working in the same abstraction level.

In Table 3.1, we introduce some terms useful for understanding how the modeling and simulation are addressed in Metropolis.

Term	Definition
Heterogeneity	Ability of a model to include processes associated to multiple domains.
Domain	Application area or discipline, e.g. multimedia, automotive, wireless communication, etc.

Table 3.1: Heterogeneity and Domain Definitions in Metropolis.

a. Modeling in Metropolis

The framework infrastructure consists, in part, of an internal representation mechanism called *Metropolis Meta Model (MMM)*, which is a set of abstract classes that can be derived to model a well-separated **computation** and **communication** semantics: it supports the notion of concurrent *processes* communicated through *ports*, *interfaces* and *mediums* (*channels*).

The metamodel semantics is powerful, it can be used for: (1) representing models at the *functional abstraction level*, (2) representing models at the *architectural abstraction level*, (3) supporting the encapsulation of *both abstraction levels* in a same network, and refining these networks and their behavior through the definition of common constraints.

(1) The ***functional abstraction level***, as shown in Figure 3.2 (a), allows the representation of models as a set of interconnected objects, which take actions while communicating with one another. These objects, called **processes**, are atomic elements describing computations as sequential programs called **threads**. They communicate through **ports**, using a set of methods declared by means of **interfaces**. As in the case of SystemC, the *interfaces* are implemented by other independent objects defined to be interconnected between *ports*, in Metropolis these objects are called **mediums**.

Using the objects previously described, the model designer can describe a network of functional processes, whose execution semantics is restricted by a set of logic formulas called **constraints**. These constraints must be specified by the designer, and they are responsible of the coordination and synchronization of processes.

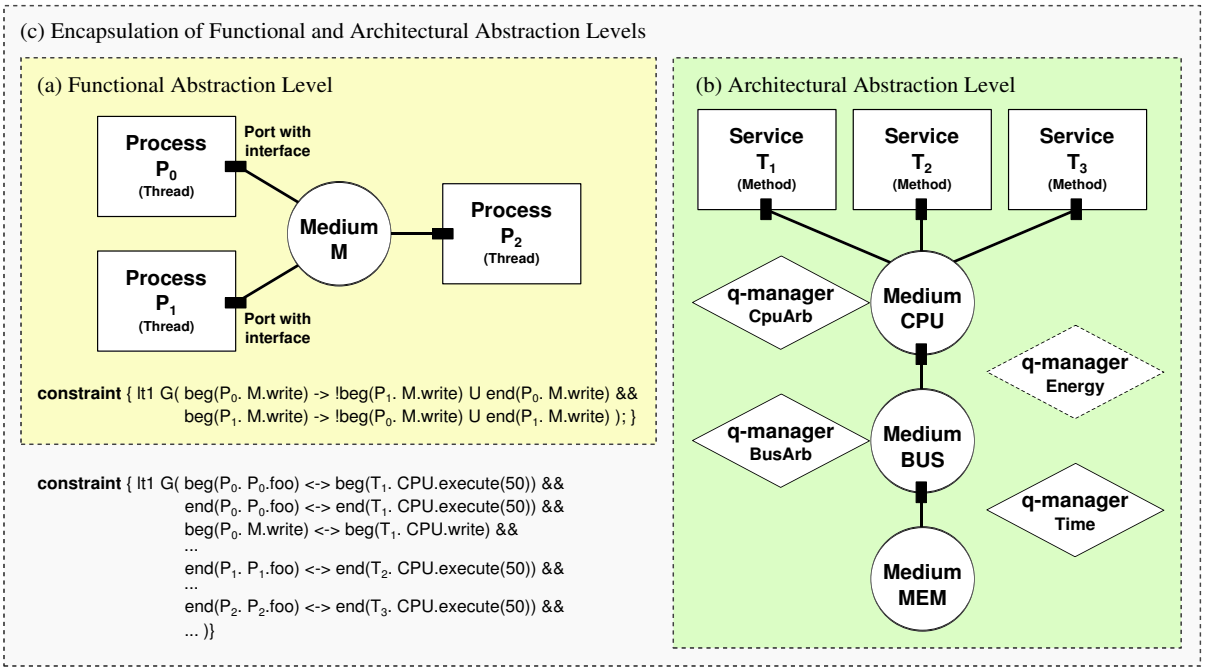

Figure 3.2: Modeling in in Metropolis (adapted from [38]).

(2) The ***architectural abstraction level***, as shown in Figure 3.2 (b), allows the representation of models based on: the *functionality* that can be modeled, and the *efficiency* with which it is modeled. **Functionality** is expressed through a set of services in the architecture: *methods* bundled to *interfaces*; and **efficiency** is expressed by the execution cost of each service, which is measured by *quantity managers*.

These **quantity managers** are responsible for controlling the execution semantics of different architectural components, they ensure the coordination of the simulation, and can be used for modeling shared architectural resources, for example: buses, CPU scheduling algorithms or simulation times. Although some quantities are available in the Metropolis framework, designers can write different ones to support specific application domains.

(3) The ***encapsulation of functional and architectural models***, as shown in Figure 3.2 (c), defines a new network, and relates the execution of all the included components by means of additional *synchronization constraints* defined by the designer. Generally, the *architectural model* should provide services at a particular cost, while the *functional model* should use these services.

b. Representation of Heterogeneity in Metropolis

Specific and separated models of computation and solvers are not defined in Metropolis. Heterogeneity can only be represented using processes, mediums, quantities and constraints. *Processes* represent the modules, whose behaviors can be associated to different domains; *mediums* allow the interactions among them; and *quantities* and *constraints* control their execution. Heterogeneity is represented at the *language-level* making the metamodel semantics to be mapped on a single simulation kernel.

c. Simulation in Metropolis

As the execution order of the processes in Metropolis models should depend only of the constraints and quantity managers implemented by the designer, the simulation semantics is based on the interaction of two netlists: a **scheduled netlist**, which contains the processes and mediums representing the system behavior; and a **scheduling netlist**, which contains the constraints and quantity managers (e.g. *q-manager Energy*, and *q-manager Time* shown in Figure 3.2), which mesure the execution costs and model the scheduling polices of a system. Two phases are performed by the Metropolis simulation kernel, as shown in Figure 3.3:

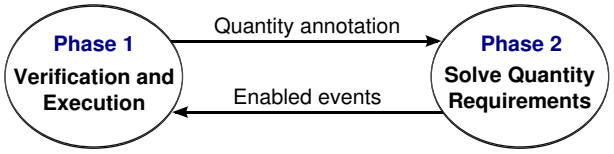

Figure 3.3: Simulation Phases in Metropolis.

- **Phase 1 – Verification and Execution**: where the *scheduled netlist* verifies the existence of events and the availability of services, if all associated conditions are satisfied, the events are executed. In this phase, quantity annotations or requirements can be generated depending on whether two processes request access to the same service. When it occurs, the next phase begins.
- **Phase 2 – Solve Quantity Requirements**: where the requirements are solved by the *scheduling netlist* and the quantities are updated. Later, the first phase is re-executed.

d. Summary of the Metropolis Important Features

- In functional models the separation between computation and communication can be compared with SystemC (SC): notion of *process* (*module* in SC), which communicates through *ports*, thanks to the methods defined by *interfaces* and implemented by *mediums* (*channels* in SC).

- Hierarchical models are not allowed: all processes should be implemented in the same hierarchical level to be interconnected using mediums.
- The model designers have the difficult task of expressing the synchronization:
  - Time synchronization by means of *constraints* and *quantity managers*.
  - Data synchronization by means of *mediums*, which can be used as converter channels when two interconnected modules represent behaviors associated to different domains.
- The framework supports a shallow heterogeneity, where only one simulation kernel handles the processes' execution.
- The metamodel does not have a predefined notion of time, but developers can model it through *quantities*.

3.2.2. Metro II

Metro II [40] is a framework created to improve the design methodology proposed by Metropolis: it adds the ability to import pre-designed IP's (described in multiple programming languages), by means of components called *wrappers*; and adds the ability to separate the model's execution costs and the scheduling policies using two different types of quantities (*annotators* and *schedulers*).

In Table 3.2, we introduce some terms useful for understanding how the modeling and simulation are addressed in Metro II.

Term	Definition
Heterogeneity	Ability of a model to include components described under multiple MoCs.
MoC	Timed or untimed computation and communication semantics, e.g. continuous time, discrete time, synchronous data flow, etc.

Table 3.2: Heterogeneity and MoC Definitions in Metro II.

a. Modeling in Metro II

In this framework, models can be implemented using different objects as shown in Figure 3.4: *components*, *ports* and *connections* for defining the specification; and *constraints*, *assertions*, *adaptors*, *annotators* and *schedulers* for controlling the execution.

- **Components**: are the blocks used to encapsulate zero or more processes, and can be related to other components through *ports*. There are two types of components: **atomic components**, where the behavior is specified in a particular language and encapsulated using *wrappers*; and **composite components**, where at least two elements (defined using the Metro II semantics) are interconnected. In the case of *atomic components*, **wrappers** are the elements specified by the designer to translate and expose the appropriate events and interfaces of a particular behavior (IP).

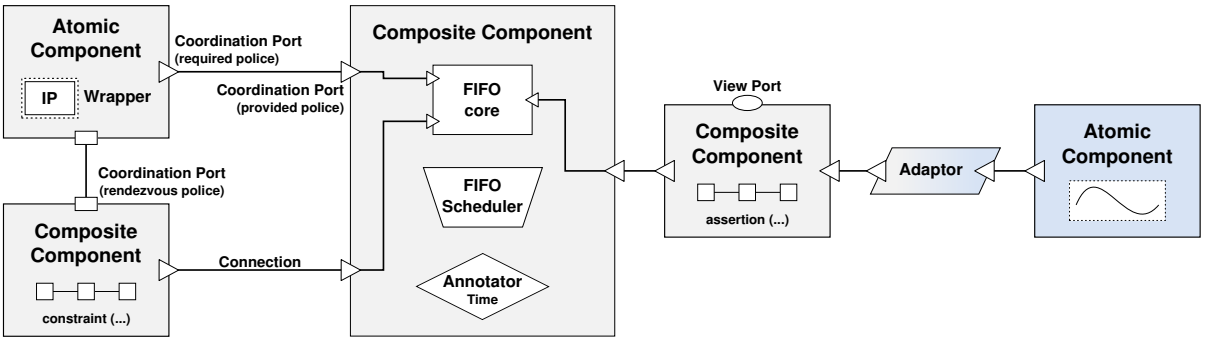

Figure 3.4: Modeling in Metro II (adapted from [40]).

- **Ports**: are the objects divided by functionality (*coordination* and *view*), which allow the communication among components. **Coordination ports**, allow an interaction of components using sequences of events (*methods*) limited by constraints. These ports can be connected to other ports, and implement different interaction policies. **View ports**, expose internal events of a component to the outside world, and cannot be connected to other ports.

- **Connections**: are the means by which the ports are interconnected.

- **Constraints and assertions**: while *constraints* are used to limit the execution of a model and specify it in a declarative form (as in Metropolis), *assertions* are used to check the execution following some restrictions during simulation. Both are declarative propositions allowing the port coordination. These objects impose restrictions for the *time synchronization* in a model.

- **Adaptors**: are the bridge between the semantics of components belonging to different MoCs, e.g. a data flow to analog adaptor can ensure the *data synchronization* among one data flow composite component and one continuous time atomic component.

- **Annotators and schedulers**: are the quantity managers implemented in Metropolis, but separated in two scenarios. *Annotators* write tags to events (for handling the model's execution costs), and *schedulers* enable or disable events (according to the scheduling polices defined). These objects collaborate with the *time synchronization*.

b. Representation of Heterogeneity in Metro II

Heterogeneity in Metro II can be mainly expressed by means of components and adaptors. *Components* describe the behaviors associated to a specific MoC; and *adaptors* allow the interaction among such MoCs. In this way, designers can express heterogeneity in the *language-level* and implement the objects responsible for ensuring the synchronization. During simulation, all the objects are mapped on a single simulation kernel, as in the Metropolis framework.

c. Simulation in Metro II

The execution of a Metro II model is based on the connection and coordination of components. It is performed in three phases, as shown in Figure 3.5:

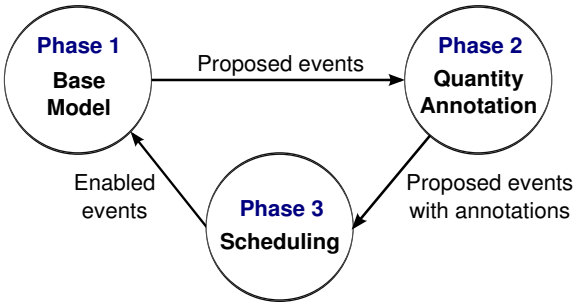

Figure 3.5: Simulation Phases in Metro II (adapted from [40]).

- **Phase 1– Base model**: where the processes defined by the designer, as a set of events (by means of components), are executed. These executions can produce new set of events that will be later consumed.
- **Phase 2 – Quantity annotation**: where each new event is associated with several quantities (annotators and schedulers).
- **Phase 3 – Scheduling**: where some events are enabled to be executed, depending on the associated annotations or scheduling polices.

d. Summary of the Metro II Important Features

- A particular level of hierarchy is allowed by the definition of *components*.
- Despite the new design objects presented in Section 3.2.2.a, the model designer still has the difficult task of implementing the synchronization according to its needs:
  - Time synchronization by means of *constraints*, *assertions*, *annotators* and *schedulers*.
  - Data synchronization using *adaptors*.
- The framework supports a shallow heterogeneity, where only one simulation kernel handles the processes execution.
- The metamodel does not have a predefined notion of time, but developers can model it through *annotators*, which handle the time for the particular services offered by the model.

3.2.3. Ptolemy II

Ptolemy II [41]–[44] is a software environment based on a *structured and hierarchical heterogeneous approach*, which focuses on the design and simulation of complex heterogeneous systems. It allows designers to formulate *homogeneous systems* capable of achieving heterogeneity by the interconnection of sub-models associated with different domains. These homogeneous systems refer to the set of interconnected components (network of *actors*) handled by a same execution and communication semantics.

In Table 3.3, we introduce some terms useful for understanding how the modeling and simulation are addressed in Ptolemy II.

Term	Definition
Heterogeneity	Ability of a model to include actors associated to multiple domains.
Domain	Implementation of a MoC.
MoC	Set of laws that govern the interactions and the execution of a model.

Table 3.3: Heterogeneity, Domain and MoC Definitions in Ptolemy II.

a. Modeling in Ptolemy II

Ptolemy II advocates an actor-oriented view of a system, as shown in Figure 3.6, where the structure is modeled by *actors* and *ports*, and the interactions are modeled by *communication channels* and *domains*.

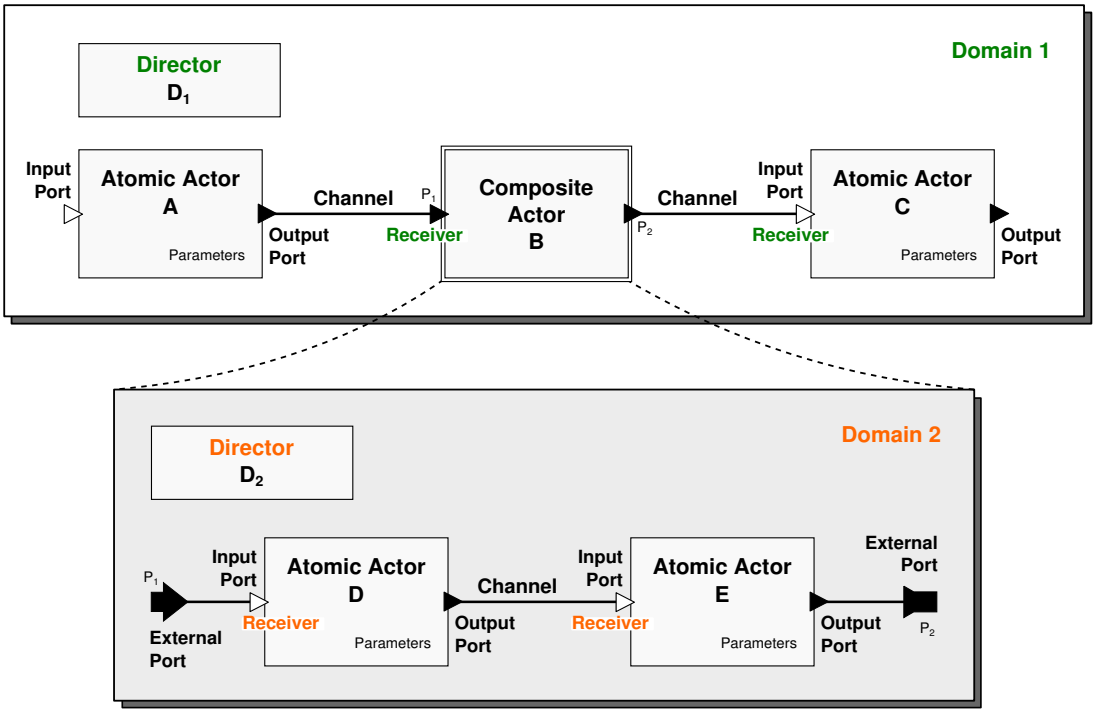

Figure 3.6: Hierarchical Modeling in Ptolemy II (adapted from [42]).

- **Actors**: are the basic concurrent blocks described in Java, and used for encapsulating a behavior associated to a particular domain. They can be separated in two types: **atomic actors**, which are described in the lowest hierarchical level; and **composite actors**, which can contain other composite or atomic actors.
- **Ports**: represent the communication points among *actors*. They are separated in three types according to their functionality: *input*, *output* and *inout ports*. When *input* and *output ports* are used to communicate between different levels of hierarchy, they are called **external ports**.
- **Communication channels**: are the explicit mechanisms used to transfer data among ports. These mechanisms are available in the framework, according to each actor *domain*. Generally, *actors* communicate through *ports* using *channels*.

- **Domains**: represent the MoC implementation associated to a composite actor. They are defined using *directors* and *receivers*. While **directors** control the execution of sub-actors belonging to the composite actor where they are instantiated, **receivers** (implemented in inputs ports) define the communication mechanism between a communication channel and a port located in the same hierarchical level. This means that *time synchronization* is handled by means of directors and *data synchronization* by means of receivers.

In a model, designers can directly instantiate a director into a composite actor to ensure that all its sub-actors will follow a particular communication semantics. When a director is not instantiated in a composite actor, it takes the semantics defined by its upper hierarchical level. In the framework, the choice of the *directors* and *receivers* to be instantiated at each hierarchical level is in the hands of the model designers.

b. Representation of Heterogeneity in Ptolemy II

In Ptolemy II, deep heterogeneity is supported by the independent definition of different domains, each of which implements a MoC. In the framework, a domain has a set of available predefined actors, ports, channels, directors (including solvers for computing the control flow of actors), and receivers for controlling how the models can be defined and how they will be executed. This means that the heterogeneity is handled at the *kernel-level*: domain-specific kernels drive the simulation following an *abstract execution semantics* imposed on the Ptolemy actors. Some examples of domains included in Ptolemy II are presented in [45].

c. Simulation in Ptolemy II

In Ptolemy II, directors handle the execution of models defining the control flow of actors and their communication semantics. It is possible thanks to an **abstract execution semantics** [43] associated to the actors. In a generic way, Ptolemy actors are executed by the phases (abstract methods) shown in Figure 3.7.

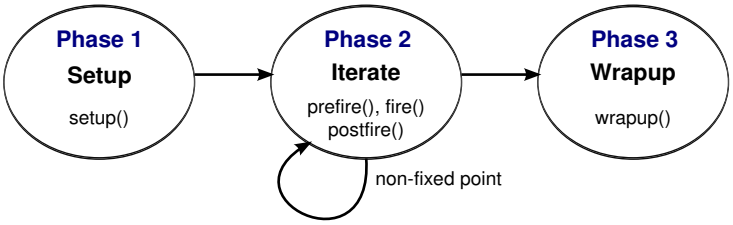

Figure 3.7: Simulation Phases in Ptolemy II.

- **Phase 1 – Setup**: is the phase in which the initialization occurs in two stages: (1) *preinitialize*, responsible of the definition of the structural required information, the dynamic construction of actors, and the receivers' creation; and (2) *initialize*, responsible of the initialization of the actor parameters, the reset of states, and the initial production of tokens associated to input or output ports.

- **Phase 2 – Iterate**: is the phase which refers to the execution of actors. In Ptolemy, the atomic executions are called *iterations* and are considered as finite computations that lead the actor to an inactive state. Specifically, in *composite actors*, the director determines how the iterations are related to the actors.

  Iterations are divided in three stages: (1) *prefire*, which tests the required conditions for executing the actor; (2) *fire*, which performs the actor computation until it reaches a fixed point in which its state remains constant (consumes the inputs, processes the inputs, and produces the outputs); (3) *postfire*, which updates the actor state.

- **Phase 3 – Wrapup**: is the phase in which the resources allocated by the actors are released.

Actors are designed assuming the definition of the abstract semantics previously introduced, not its specific implementation, because it is provided by the model of computation where each actor is embedded. This means that simulation phases are implemented by the directors instantiated at each particular hierarchical level. For example, in the model hierarchy shown in Figure 3.6, the director $\mathbf{D_1}$ implements the actions to be performed when the setup(), prefire(), fire(), postfire() and wrapup() functions are called on the **A** and **C** actors; and the director $\mathbf{D_2}$ implements the actions to be performed, when the same abstract functions are called on the **B** actor.

d. Summary of the Ptolemy II Important Features

- Composite actors provide the notion of hierarchy, which is the most powerful feature of Ptolemy.
- The domain's definition is composed by a set of predefined elements: *actors*, *ports*, *channels*, *directors* and *receivers*. Designers can use these elements for creating their models.
- The framework supports a deep heterogeneity, where multiple kernels control the simulation.
- Model designers have the task of instantiating the elements, which handle the synchronization:
  - Time synchronization by means of *directors*.
  - Data synchronization by means of *receivers*.
- As synchronization can be handled at different hierarchical levels, and each hierarchical level can represent a particular domain: hierarchical synchronization methods control the execution of a model.
- The notion of time in a composite actor is handled by the instantiated director: it always follows the time notions imposed by the director instantiated in the upper hierarchical level.
- The predefined directors and receivers implement the semantics for interfacing two different domains. Although several directors are available for a particular domain in Ptolemy II, only one can be instantiated by level of hierarchy.
- Simulation's execution is controlled by means of the abstract semantics associated to each actor in the framework.

3.2.4. Preliminary Conclusions

Having analyzed the simulation frameworks presented in Section 3.2, and summarizing them in Table 3.4, we can conclude that:

- A multi-domain simulation framework should offer:
  - The means for supporting **hierarchical modeling** because it ensures a higher level of expressiveness.
  - **Predefined and independent elements** for ensuring the *time synchronization* and *data synchronization* between model components belonging to different timed or untimed domains. In this way, designers avoid the difficult task of expressing the synchronization at the language-level.
- The heterogeneity implemented at the **kernel-level** allows a better separation among the different domains included in a framework, because each domain can be implemented and simulated using a specific kernel.
- The approach to hierarchically handle the simulation time and synchronization among timed or untimed domains is a powerful feature, which can reduce the complexity when simulating multi-disciplinary models.

  Assuming a model implemented in two hierarchical levels, where each level represents a different domain, interactions between such hierarchical levels are simplified into a **master-slave relation**: the master domain (implemented at the higher hierarchical level) imposes the time or synchronization semantics to be followed by the slave domain (implemented at the lower hierarchical level).
- The **abstract semantics** provided by Ptolemy II introduces the principles for a generic simulation and synchronization of components described under different timed or untimed domains.

3.3. Frameworks Specified Using SystemC

3.3.1. HetSC

HetSC [46], [47] is a framework for the specification and design of concurrent heterogeneous embedded systems in SystemC. Its objective is to allow the designer to express heterogeneity based on the SystemC primitives; and propose mechanisms to include and interconnect within the same environment, processes belonging to different MoCs.

In Table 3.5, we introduce some terms useful for understanding how the modeling and simulation are addressed in HetSC.

HetSC is proposed in two levels described by means of a *general specification methodology* and a *heterogeneous specification*, as shown in Figure 3.8.

Framework	Metropolis	Metro II	Ptolemy II
Hierarchical Modeling	It is not allowed.	It is allowed in one level by means of the ability to import pre-designed IP.	It is allowed in multiple levels by means of *composite actors*.
Separation between Computation and Communication	- Computation by means of *processes*. - Communication by means of *ports*, *interfaces* and *mediums*.	- Computation by means of *components*. - Communication by means of *ports* and *connections*.	- Computation by means of *actors*. - Communication by means of *ports* and *channels*.
Heterogeneity Level	Language-level.	Language-level.	Kernel-level.
Synchronization	- Time synchronization imposed by means of *constraints* and handled by *quantity managers*. - Data synchronization handled by means of *mediums*.	- Time synchronization imposed by means of *constraints* and *assertions*, and handled by *annotators* and *schedulers*. - Data synchronization handled by means of *adaptors*.	- Time synchronization handled by means of *directors*. - Data synchronization handled by means of *receivers*.
Time Notion	A global time notion is handled by means of *quantities*.	A global time notion is handled by means of *annotators*.	A distributed time notion is handled by the *directors* instantiated at each hierarchical level.
Advantages	Good separation between computation and communication.	- Good separation between computation and communication. - Hierarchical modeling introduced.	- Good separation between computation and communication. - Full hierarchical modeling. - Hierarchical approaches for handling synchronization and local time notions in each domain. - Domains are well-separated. - Abstract semantics for controlling simulation.
Disadvantages	- Missing hierarchical modeling. - Synchronization and global time notion handled by the designer at the language-level. - Domains are not well separated.	- Synchronization and global time notion handled by the designer at the language-level. - MoCs are not well separated.	Instantiation of elements controlling the time synchronization in each hierarchical level (*directors*) is the responsibility of designers.

Table 3.4: Summary of Features of Frameworks Based on Metamodels and High-Level Programming Languages.

Term	Definition
Heterogeneity	Ability of the framework to support and integrate several MoCs in a same specification.
MoC	Primitives and specification rules for describing the characteristics of processes and the interactions among them.

Table 3.5: Heterogeneity and MoC Definitions in HetSC.

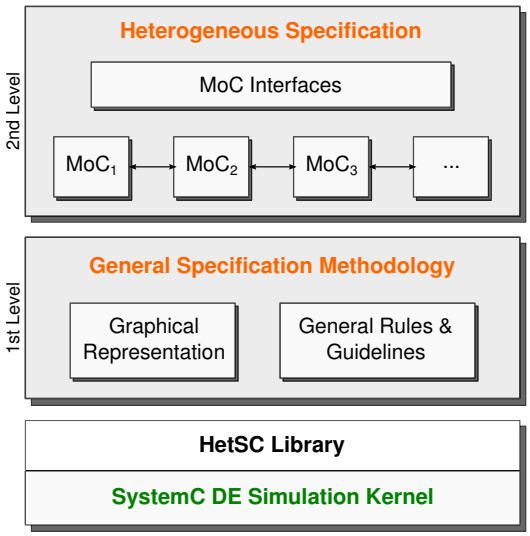

Figure 3.8: Architecture of the HetSC Framework (adapted from [47]).

a. Modeling in HetSC

The **HetSC general specification methodology** includes the *graphical representation* of SystemC constructs, and defines the set of *rules and guidelines* imposed for the specification of concurrent systems. A typical HetSC specification is shown in Figure 3.9.

- **Graphical representation** is the set of graphical symbols used for developing a model.
- **Rules and guidelines** are the means by which a system is separated from its environment.

In the specification methodology several hierarchical levels can be implemented. The top-level instantiates a sc_main() function containing the model parts: (1) the *environment*, which provides stimuli and checks restrictions; and (2) the *system*, which encloses the definition of modules, ports, interfaces and channels in different hierarchical levels.

HetSC follows the same SystemC approach where the computation (represented by processes) is well-separated from the communication (represented by ports, interfaces and channels). This means that the only way to communicate *processes* is through *channels*. For this reason, **channels** involve the semantics for handling the synchronization among two or more processes belonging to the same model of computation: they can block or unblock the processes' execution.

The framework includes a library of **predefined channels** that can be instantiated for communicating several processes. Designers need to know the semantics and syntax of each predefined channel to instantiate and access them from the processes. Additional channels can be also defined by a designer according to their needs.

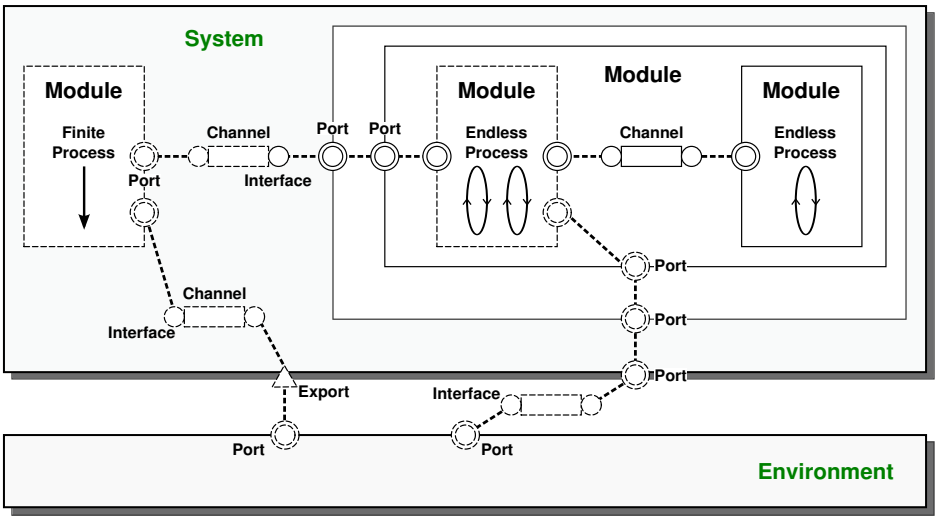

Figure 3.9: Modeling in HetSC: Specification Primitives (adapted from [46]).

b. Representation of Heterogeneity in HetSC

The **HetSC Heterogeneous Specification** supports the implementation of different MoCs at the *language-level*. It handles the MoC specification as a mechanism (rules and guidelines) to construct models, and defines the *MoC interfaces* allowing the communication and interaction between different MoCs. Some examples of MoCs implemented in HetSC are presented in [48].

MoC interfaces are special *border processes* and *border channels* which connect processes described under different MoCs. They should implement a set of language primitives responsible for the interactions among MoCs. A representation of these interfaces is shown in Figure 3.10.

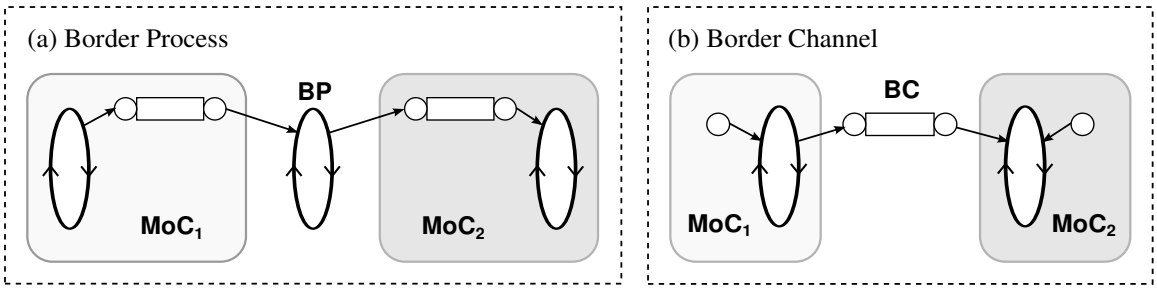

Figure 3.10: MoC Interfaces in HetSC (adapted from [46]).

- **Border Processes**: where the designer adds the code for adapting the interaction among channels, defined under two different MoCs. It is quite flexible because even in the predefined border processes, offered by the HetSC library, designers can modify the code for adapting the interactions. However, it is difficult because the designer should guarantee the synchronization among the channels bound to the border process.
- **Border Channels**: where the adaptation of semantics among MoCs is provided. Designers cannot modify them.

c. Summary of the HetSC Important Features

- *Modules*, *ports*, *interfaces*, and *channels* provide the notion of hierarchy.
- The framework supports a shallow heterogeneity, where all synchronization methods and MoC definitions are handled at the language-level; and are mapped on the SystemC DE kernel for ensuring simulation.
- Designers have the task of instantiating or implementing the elements, which handle the interaction and synchronization:
  - **Channels**: link processes described under the same MoC.
  - **MoC Interfaces**: synchronize processes described under different MoCs.
- Time and data synchronization methods are not separated in two independent elements. They have to be implemented by the designers according to its needs.
- The notion of time is handled by the SystemC DE simulation kernel, but some considerations needed for the MoC operation are implemented through channels, which provide the communication semantics between MoCs.

3.3.2. HetMoC

HetMoC [49] is a framework in SystemC for the specification and simulation of heterogeneous distributed systems. It is based on a formal base, which offers a clear separation between computation and communication.

In Table 3.6, we introduce some terms useful for understanding how the modeling and simulation are addressed in HetMoC.

Term	Definition
Heterogeneity	Ability of a model to support and integrate processes, signals and interfaces described under different MoC Domains.
MoC Domain	Continuous time, discrete time, synchronous/reactive or untimed semantics used for describing process in a network.

Table 3.6: Heterogeneity and MoC Domain Definitions in HetMoC.

a. Modeling in HetMoC

HetMoC Models, as shown in Figure 3.11, are represented as a set of *processes* and *signals*, which can be grouped into different MoC domains through elements called *domain interfaces*.

- **Processes**: specify the computation through a function, mapping input signals to output signals. In the framework, they are fully implemented by the designer using SystemC threads which, at the same time, are encapsulated into SystemC modules.

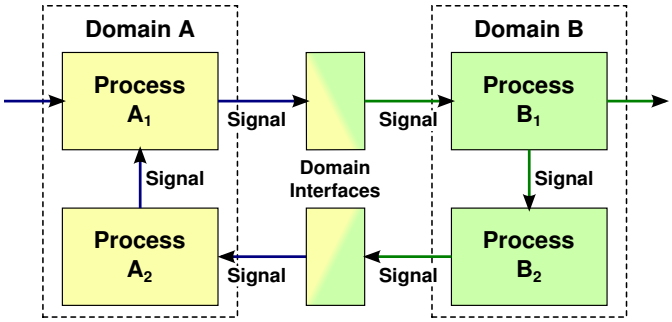

Figure 3.11: Modeling in HetMoC: Processes, Signals and Domain Interfaces (adapted from [49]).

- **Signals**: are sequences of data events containing the information about the time abstraction level allowed in each MoC domain. Specific signals are predefined in the framework (e.g. continuous time, discrete event, data flow, and synchronous/reactive signals). These signals offer to the designer the means for linking their processes.
- **Domain interfaces**: are the elements responsible for transferring the signal information across different MoC domains. They are functional mapping functions, which preserve the causality and monotonicity of the involved signals.

In this approach, the designer implements the functionality of processes by means of SystemC threads, and instantiates the *predefined signals* and *domain interfaces* offered by the HetMoC framework. The objective of the designer is to correctly relate their processes through the available predefined communication elements. Some examples are presented in [49].

b. Representation of Heterogeneity and Simulation in HetMoC

Heterogeneity is implemented at the *language-level*. By means of SystemC primitives, *processes*, *signals* and *domain interfaces* are defined and mapped on the SystemC DE kernel for performing the simulation. Unfortunately, details about the simulation semantics are not provided.

c. Summary of the HetMoC Important Features

- Hierarchical models are not allowed.
- The framework supports a shallow heterogeneity, where all modeling elements are defined using SystemC primitives and are simulated under a DE simulation kernel.
- Designers have the task of connecting their processes through predefined elements, which handle the interaction and synchronization:
  - **Signals**: link processes described under the same MoC Domain.
  - **Domain interfaces**: synchronize processes described under different MoC Domains.

3.3.3. ForSyDe

ForSyDe [50]–[52] is a specification framework enabling the modeling and simulation of heterogeneous embedded systems. It is implemented as a C++-based class library on top of the SystemC standard, it reuses the SystemC DE simulation kernel and defines new modeling elements based on the SystemC primitives.

In Table 3.9, we introduce some terms useful for understanding how the modeling and simulation are addressed in ForSyDe.

Term	Definition
Heterogeneity	Ability of a model to support several MoCs.
MoC	Describes the semantics of computation and concurrency, and models the time abstraction of each process of a model.

Table 3.7: Heterogeneity and MoC Definitions in ForSyDe.

a. Modeling in ForSyDe

In ForSyDe, a system model separates computation from communication; and follows particular semantics [53], which can be executed using functional or high-level programing languages. This system model, as shown in Figure 3.12, is represented as a set of concurrent hierarchical process networks, which is integrated by *processes* and *domain interfaces* connected through *signals*. This approach improves the one presented in Section 3.3.2.

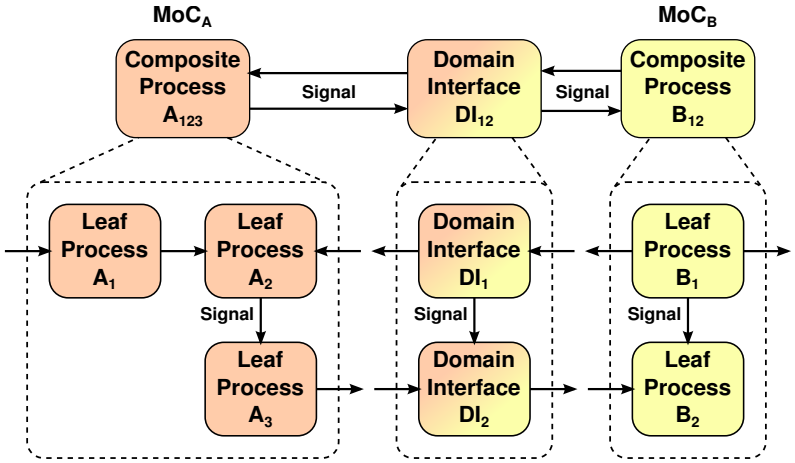

Figure 3.12: Modeling in ForSyDe: Processes, Signals and Domain Interfaces (adapted from [50]).

- **Processes**: are functional objects defined by the designer to receive input tokens, invoke a function (defined by a constructor), and produce and communicate the output tokens to other processes. In the framework implementation, processes are realized by means of SystemC modules, which invoke functions provided by the designer.

  Processes can be classified in two types: **composite processes**, which are created by composing other processes together; and **leaf processes** created using *process constructors*, which are pre-

defined constructors available in a ForSyDe library. These predefined constructors ensure the computation and communication between processes.

- **Domain Interfaces**: are particular *processes* instantiated for allowing the connection between different models of computation: they should define the synchronization interface among processes belonging to different MoCs. In ForSyDe, a MoC is used to model the timing abstraction of processes.
- **Signals**: are the mechanisms used to communicate. They are considered as set of events conveying data tokens among processes. In the framework's implementation, signals are mapped to SystemC FIFO channels.

b. Representation of Heterogeneity in ForSyDe

ForSyDe supports the implementation of different MoCs at the *language-level*: the constructors of each MoC (SystemC module-based classes) are implemented based on an *abstract simulation semantics* (similar to the Ptolemy semantics), and they are mapped onto a single *simulation model*, which uses blocking writes to bounded FIFOS [50]. This simulation model control the simulation execution.

c. Simulation in ForSyDe

The **simulation model** in ForSyDe is based on the **abstract simulation semantics** presented in Figure 3.13. This semantics is similar to the one presented in Section 3.2.3.

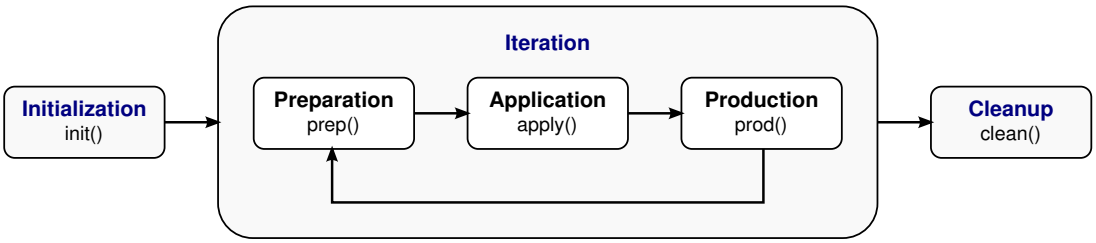

Figure 3.13: Simulation Phases in ForSyDe (adapted from [50]).

- **Initialization**: where the memory allocations and the initialization of variables are performed.
- **Iteration**: where the next three stages are repeated until they consume all inputs, reach a deadlock or find the interruption of a process.
  - **Preparation**: where the process prepares or updates its inputs.
  - **Application**: where a function is provided to generate its outputs.
  - **Production**: where the synchronization with the system kernel occurs, and the correct number of tokens are written to the output ports.
- **Cleanup**: where the resources allocated during execution are released.

The specific implementation of each stage in ForSyDe is provided by the definition of each MoC. This means that different implementations of each init(), prep(), apply(), prod() and clean() functions are available according to the process instantiated in a model. An example of how the abstract semantics is implemented for different MoCs in ForSyDe is presented in [50].

d. Summary of the ForSyDe Important Features

- The notion of hierarchy is provided by means of *composite processes*.
- The framework supports a shallow heterogeneity, where the synchronization methods are handled inside each particular MoC definition, and later they are mapped upon a single simulation model, which controls the simulation's execution.
- Designers have the task of instantiating and connecting the elements, which handle the interactions and synchronization among processes:
  - **Signals**: link processes described under the same MoC.
  - **Domain interfaces**: synchronize processes described under different MoCs.
- The notion of time is encapsulated in the process constructors associated with the different MoCs allowed in the framework.
- Simulation is handled by means of abstract semantics associated to processes in the framework.

3.3.4. Preliminary Conclusions

Having analyzed the simulation frameworks presented in Section 3.3, and summarizing them in Table 3.8, we can conclude that:

- **SystemC-based approaches** offer a prominent separation among communication and computation.
- Providing heterogeneous modeling based only in the DE kernel is not easy: only shallow heterogeneity is supported, and the simulation of processes is not well-separated by MoC.
- Modeling can be simplified by the separation of elements responsible for handling the interactions among processes described under the same timing abstraction (*channels* or *signals*), from the elements responsible for handling the interaction and synchronization among processes described under different timing abstractions (*MOC interfaces* or *domain interfaces*).
- Offering **predefined elements for handling the synchronization** is a powerful approach, but it leaves to the designer the responsibility of instantiating them at each hierarchical level, and this can become a complicated task.
- **Abstract simulation semantics** provides excellent means for separating and controlling synchronization at different levels of hierarchy.

Framework	HetSC	HetMoC	ForSyDe
Hierarchical Modeling	It is allowed in multiple levels by means of SystemC components.	It is not allowed.	It is allowed in multiple levels by means of *processes* (SystemC modules).
Separation between Computation and Communication	- Computation by means of *processes*. - Communication by means of *ports*, *interfaces* and *channels*.	- Computation by means of *processes*. - Communication by means of *signals*.	- Computation by means of *processes*. - Communication by means of *signals*.
Heterogeneity Level	Language-level.	Language-level.	Language-level.
Synchronization	By means of *MoC interfaces*.	By means of *domain interfaces*.	By means of *domain interfaces*.
Time Notion	- Global time handled by the SystemC DE kernel. - Particular time restrictions implemented through channels defined in each MoC.	Non-provided information.	Encapsulated in the process constructors defined inside each MoC, and mapped on the single simulation kernel.
Advantages	- Separation between computation and communication. - Hierarchical modeling. - Global time notion handled by the DE kernel. - Library of predefined elements for controlling the synchronization. - Interactions among domains handled by special elements (*MoC interfaces*).	- Separation between computation and communication. - Interaction among domains is handled by means of special elements (*domain interfaces*).	- Separation between computation and communication. - Hierarchical modeling. - Notion of time handled by each MoC and mapped on a single simulation kernel. - Interaction among domains is handled inside each MoC definition, by means of special elements (*domain interfaces*). - Abstract simulation semantics.
Disadvantages	- Instantiation of elements controlling the synchronization (*MoC interfaces*) can be implemented or re-defined by the designer.	- Missing hierarchical modeling. - Time notions are not clearly defined. - Elements controlling the synchronization must be instantiated by the designer.	- Selection and instantiation of elements controlling the synchronization (*domain interfaces*) are the responsibility of the designer.

Table 3.8: Summary of Features of Frameworks Specified Using SystemC.

3.4. Frameworks Extending the SystemC Discrete Event (DE) Simulation Kernel

3.4.1. SystemC-H

SystemC-H [54]–[56] is a simulation prototype created for the heterogeneous modeling and simulation in SystemC. It is an approach, which extends and modifies the SystemC simulation kernel by adding three specific and separated MoCs: Synchronous Data Flow (SDF), Finite State Machine (FSM), and Communicating Sequential Processes (CSP).

In Table 3.9, we introduce some terms useful for understanding how the modeling and simulation are addressed in SystemC-H.

Term	Definition
Heterogeneity	Ability of the framework to integrate several MoCs.
MoC	Set of constructors and process composition operators, which provide timing semantics for a model.

Table 3.9: Heterogeneity and MoC Definitions in SystemC-H.

a. Modeling in SystemC-H

SystemC-H allows the description of models in only one hierarchical level and does not provide synchronization mechanisms among components, which are described by means of the different implemented MoCs.

In addition, the framework does not have a global modeling approach regardless of the MoC to be used. This means that the components of a model, parameters, means of connection among them, and implementation are imposed by a set of guidelines specific to each MoC, instead of inheriting the SystemC predefined components. Some examples of MoCs included in SystemC-H are presented in [55].

In this section, we do not analyze the MoC-specific modeling guidelines, because we are interested in proposing a generic simulation approach, where a model remains integrated by SystemC components, or derived from them.

b. Representation of Heterogeneity in SystemC-H

The heterogeneity in SystemC-H is supported by the independent definition of different MoCs: each one with a set of classes, which provide the specific modeling and simulation semantics associated to each domain. This means that the heterogeneity is handled at the *kernel-level*: MoC-specific kernels handle the simulation of processes.

c. Simulation in SystemC-H

Execution of models is based on a one-level **master-slave relation**, where a *DE modified kernel* supports the initialization and simulation of processes described by means of different *MoC-specific*

kernels. In this approach the *SystemC scheduler*, introduced in Section 2.2.2, preserves the *initialization*, *evaluate*, *update*, *delta* and *time notification phases*, but some of them are modified as follows:

- **Separation of initialization roles**: although the processes are implemented under different MoCs, they remain *SystemC methods*, which are registered in the DE kernel to be executed once during the *initialization* scheduler phase. It is not desired for some MoCs, as is the case of the implemented SDF MoC, because the order of execution of their processes can be clearly specified before *simulation*. For this reason, the framework includes in the DE kernel one function for splitting the SDF processes from the regular SystemC method processes.
- **Specification of an execution order**: according to the framework approach, when the designers can determine by means of a MoC-specific kernel the execution order of their processes, they should have the possibility of forcing the DE kernel to respect it. For this reason, the function that executes all the SystemC processes in the scheduler is altered.
- **Control of MoC processes' execution according to the DE time**: as MoC-specific kernels can require particular time scales for executing their processes, a variable is added in the SystemC simulate() function for monitoring the edges of the SystemC clock.

The *DE modified kernel* proposed for handling the simulation of models in the framework was defined only based on particular MoCs implementations. This implies that the addition of new MoCs will probably involve the modification of the SystemC DE simulation kernel.

d. Summary of the SystemC-H Important Features

- Hierarchical modeling is not allowed.
- The framework supports a deep heterogeneity because several simulation kernels control the simulation.
- The SystemC DE simulation kernel is modified.
- Synchronization mechanisms are not available among components defined under different models of computation.

3.4.2. SystemC-A

SystemC-A [57], [58] proposes an extended version of SystemC for allowing the modeling of AMS systems at different abstraction levels. It is proposed as an alternative to SystemC-AMS, which supports *analog system variables* and *components* that can be combined to automatically generate non-linear Ordinary Differential and Algebraic Equations (ODAEs) or Partial Differential Equations (PDEs).

a. Modeling in SystemC-A

In the SystemC extension, a system can be modeled by means of two types of elements: **analog system variables** as nodes, quantities, flows, efforts or partial quantities; and **analog components**

as capacitors, resistors, voltage and current sources. The designer can interconnect these elements to define analog circuits using *netlists*, similar approach to the implemented by the ELN MoC of SystemC-AMS [13].

Although some predefined elements are provided by the extensions, the designer can define new variables, implement components, or modify them according to its needs. These modifications are implemented into a build() function, which specifies the analogue behavior of a component.

Elements in SystemC-A are implemented as a set of independent-SystemC classes sharing the base classes, which define the *abstract semantics* to be used by the *variables* and *components* during the analog simulation. The build() function is part of this abstract semantics, it is defined by a *component base class* and implemented into each SystemC-A component. Some examples of models implemented using SystemC-A are presented in [59], [60].

b. Representation of Heterogeneity in SystemC-A

SystemC-A has a specific simulation kernel responsible for handling the analog components of a model. For this reason we consider that it follows a deep heterogeneity approach. Thanks to some language constructs, the system is specified by the designer, and later constructed and simulated in the hands of an independent analog kernel, which is able to synchronize with the SystemC DE simulation kernel.

c. Simulation in SystemC-A

The digital/analog simulation in this SystemC extension involves one change in the SystemC scheduler introduced in Section 2.2.2. As shown in Figure 3.14, the change implemented in the scheduler is a call to the analog kernel phases (*iteration* and *verification*) before the SystemC scheduler *evaluation phase*.

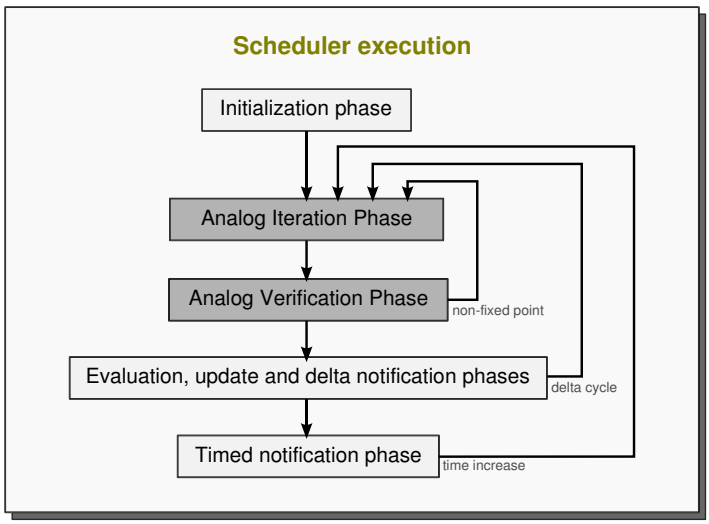

Figure 3.14: Changes Involved in the SystemC DE Kernel for enabling the Analog Simulation.

- **Analog Iteration Phase**: where the analog elements and components are initialized, and scanned to build the linearized models; then these models are solved and the solutions are updated. If the solutions converge, the *analog verification phase* is executed; otherwise the *analog iteration phase* is re-executed.
- **Analog Verification Phase**: where the analog kernel calculates the time step sizes to be used by the DE simulator. It advances until the current DE simulation time, schedules an event at a time equal to the current simulation time plus the next selected time step, and then, it suspends.

More details about simulation are presented in [57], [58].

d. Summary of the SystemC-A Important Features

- Hierarchical modeling is not allowed.
- The extension is implemented following a deep heterogeneity approach: an independent analog kernel control the analog simulation.
- SystemC DE simulation kernel is modified.

3.4.3. Preliminary Conclusions

The approaches presented in this section, provide a further step towards the creation of multi-disciplinary simulators: they propose different ways for adding models of computation on the SystemC DE simulator kernel. Despite this, they present some features that we want to avoid:

- Hierarchical modeling unsupported.
- SystemC objects not considered for defining other MoC-specific components.
- SystemC DE kernel modified according to the constraints imposed by the included MoC.
- Synchronization mechanisms are not available among components defined under different domains (excluding DE).

We seek to add some extensions to SystemC, through a generic method, without altering the simulation cycle defined by the standard, and without depending of the MoC to be included. We believe that preserving the **MoC components as SystemC objects**, and exploring the **SystemC object hierarchy** provided by the kernel during elaboration, are means to reach our goal.

3.5. Conclusion and Outlook

Having analyzed how the modeling, heterogeneity and simulation are addressed in the different simulation approaches presented in this chapter, we can identify the existing means for ensuring the *multi-disciplinary synchronization* inside the same simulation environment, and we can define the *features* that an environment should have to ensure such synchronization means.

On the one hand, we identify that for ensuring the **multi-disciplinary synchronization**, the frameworks do not propose neither a single method to successfully convert the information transmitted

among components belonging to different domains, nor a single method to ensure that the same information is transmitted in the right time. This means that for each specific pair of domains that want to interact within the same simulation environment, a method is defined for ensuring the data synchronization, and another one for ensuring the time synchronization. For example, the *data synchronization* is ensured by elements always connected among components belonging to a pair of domains (in Metro II by adaptors; in Ptolemy II by receivers; and in HetSC, HetMoC and ForSyDe by domain interfaces or MoC interfaces); and the *time synchronization* is ensured by the language-level definitions or the elements implemented for handling the time constraints among a pair of domains (in Metropolis and Metro II by constraints; in Ptolemy II by directors; and in HetSC, HetMoC and ForSyDe by domain interfaces or MoC interfaces).

The frameworks introduced in this section differ from the SystemC-AMS proof-of-concept, introduced in section 2.3.1, which proposes a unique synchronization method that is shared by all their included models of computation. This synchronization method, implemented under a discrete time semantics, is the one specified by the *SystemC-AMS synchronization layer*.

We have a particular interest to evaluate whether the proposed SystemC-AMS synchronization method is really sufficient to synchronize any pair of domains. Therefore, in Chapter 4, we first analyze such synchronization method, we introduce a formalization of the synchronization problem and a new algorithm to support it. Then, we discuss if it can be preserved as a unique synchronization method to be included into a multi-disciplinary simulation framework.

On the other hand, we define the **features**, that in our opinion, should define a true multi-disciplinary simulation environment. These features, presented below, will be considered for the definition of the proof-of-concept introduced in Chapter 5.

- Supporting *hierarchical modeling* to ensure a high-level of expressiveness.
- Supporting *heterogeneity at the kernel-level* for allowing the separation, in terms of synchronization methods, among the different domains included in the simulator framework.
- Implementing a *master-slave relation* among the models of computation for controlling the hierarchical synchronization and simulation among different domains.
- Having *predefined and well-separated elements* for ensuring that the time and data synchronization of a model is not the responsibility of the designer.
- Proposing *abstract semantics* for allowing the generic simulation of the model's components.
- Following a *SystemC-based approach*, without modifying the SystemC DE kernel and implementing the MoC components as inherited-SystemC components.

CHAPTER

4 Synchronization between the Discrete Event (DE) and Discrete Time (DT) Domains

Contents

4.1. Introduction

Before proposing the synchronization principles that will be used for defining a multi-disciplinary simulation environment, in this chapter, we carefully analyze the only synchronization method included in SystemC-AMS: we formalize, improve and evaluate it to know if it is able to ensure interactions among several domains.

This synchronization method is the one defined between the SystemC Discrete Event (DE) simulation kernel (introduced in Section 2.2.2 and described in the DE domain), and the Timed Data Flow (TDF) Model of Computation (MoC) (introduced in Section 2.3.2, and described in the Discrete Time (DT) domain).

Due to the lack of documentation analyzing how the time notions are handled in the SystemC-AMS TDF MoC during simulation, and how the time synchronization is performed between the SystemC DE kernel and the TDF MoC, we dedicate part of this chapter to such analysis.

In Section 4.2, we present the synchronization problems that may arise when TDF models are connected to models described in the DE domain. In order to provide a good understanding level, we demystify the semantics used for handling the TDF time: we identify the different time notions involved in simulations, which instantiate components described by means of DE and TDF MoCs; and we clarify how these time notions are related.

In Section 4.3, we introduce an approach to represent the TDF models and their interactions with the DE time domain. This approach is based on a formalism called Coloured Petri Nets (CPN), and is defined by means of a set of rules allowing the creation of equivalent models, which can be subsequently analyzed.

In Section 4.4, we propose an analysis method for equivalent CPN models, which allows to determine the causality of such models in regard to the DE domain. This analysis method, in the case of causal models, determines valid schedules including not only the order in which the TDF models are executed, but also the order in which their interactions with the DE domain are performed. Otherwise, it proposes model changes to fix the detected causality problems.

Finally, in Section 4.5, the chapter concludes on interactions between several domains.

4.2. Discrete Event (DE) and Timed Data Flow (TDF) Synchronization Issues

The SystemC-AMS TDF MoC is described by means of a DT particular semantics ensuring a data flow evenly distributed in time. This semantics includes timing information, which should be handled during simulation; and which should be synchronized with the timing information handled by the SystemC DE kernel.

Synchronizations involved during the DE-TDF simulations sometimes generate unwanted timing issues, which can corrupt the models' causality. In order to understand how and when these timing issues appear, we need to know how the time is handled in the TDF MoC, more specifically, how such time is handled inside each TDF module or port instantiated by the designer in a model.

4.2.1. TDF Time Management

As introduced in Section 2.3.2, the TDF modules and ports instantiated in a model have an attribute called *timestep*. This attribute should be assigned by the designer at least once inside each TDF cluster, either in a module or a port object, and later be automatically propagated (during the elaboration phase) to the remaining modules or ports that do not contain it.

To formalize the synchronization mechanism between the DE kernel and TDF MoC, we introduce the notion of *timescale*, which is associated either to a module or a port. On the one hand, in the case of modules, the timescale is responsible for governing the time instants in which the module's actions (embedded in a processing() function) are performed. On the other hand, in the case of ports, the timescale is responsible for determining which samples should be consumed or produced during each module execution.

In order to explain in detail how each timescale is handled inside a TDF model, we introduce the example shown in Figure 4.1. For the examples presented in this section, as we are interested in analyzing the TDF-DE synchronization issues, we assume that the attributes of each TDF module (*Tm*), the attributes of each TDF port (*Tp*, *R* and *D*), the number of times that each module should be executed (*q*) within a cluster period, the cluster period (*Tcls*), and the schedule (containing the execution order of TDF modules), have been previously determined.

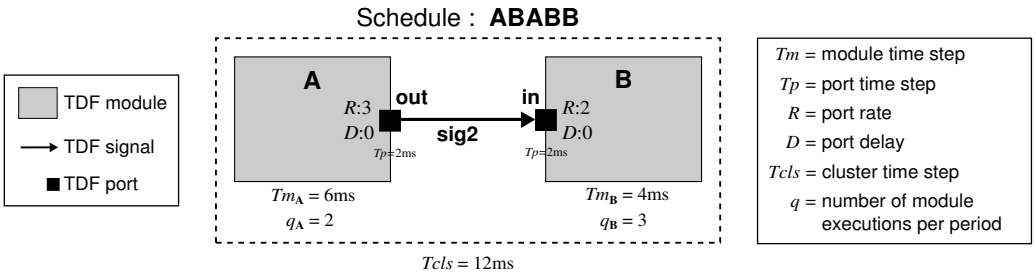

Figure 4.1: Example of a Basic TDF Cluster Composed by 2 TDF Modules and 1 TDF Signal.

a. Time Management in TDF Modules

In SystemC-AMS each module **M** has a timestep Tm_{M} indicating the time period in which its processing() function is executed. This means that the timescale associated to each TDF module progresses in time according to its own timestep.

For the example shown in Figure 4.1, during a time period of 12 ms (*Tcls*), the module **A** is executed every 6 ms (Tm_{A}): initially, the timescale of module **A** is initialized at 0 ms, when its processing() function is executed for the first time. Later, the same timescale progresses to 6 ms, to execute a second time its processing() function.

Similarly, the module **B** is executed every 4 ms (Tm_{B}): initially, the timescale of module **B** is initialized at 0 ms, when its processing() function is executed for the first time; later the timescale progresses to 4 ms, to execute a second time the same function; and finally, the timescale progresses to 8 ms, to execute a third time the same processing() function.

As the timescales of each module are independent from each other, the only condition considered to indicate the progress of such timescales is the schedule determined during elaboration. For the

example shown in Figure 4.1, the relation between the execution order of the TDF modules (**ABABB**), and the progress of each TDF timescale is illustrated in Figure 4.2.

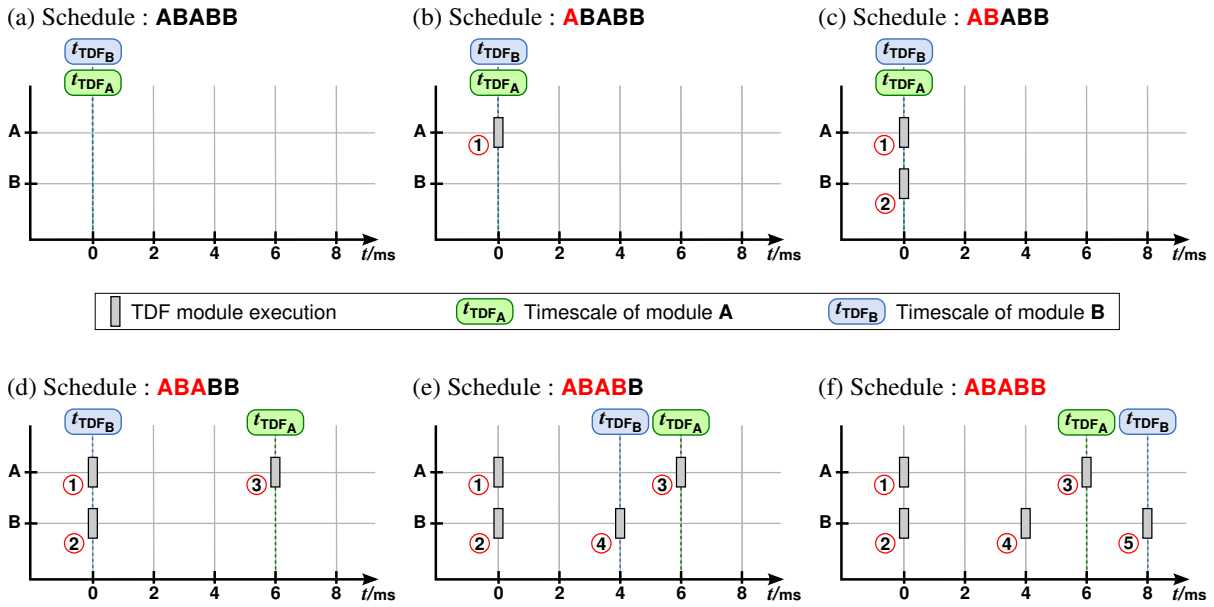

Figure 4.2: Time Management in TDF Modules Belonging to TDF Cluster shown in Figure 4.1.

b. Time Management in TDF Ports

When the processing() function of a TDF module is executed, a fixed number of samples are read from its input ports, and another fixed number of samples are written on its output ports. These numbers of samples are determined by the rate attributes associated to each port. In the example shown in Figure 4.1, when the **A** module's processing() function is executed, three samples are produced on its **A.out** output port; and when the **B** module's processing() function is executed, two samples are consumed from its **B.in** input port.

The particularity of the read and written samples is that they are annotated with a *time stamp*, which indicates their relative temporal position with respect to the local time of the consumer or producer TDF module.

In SystemC-AMS the timestep assigned to a port determines the time period with which the samples are annotated in such port. This means that the timescale associated to each TDF port progresses according to its own timestep.

During simulation, based on the attributes assigned by the designer (*Tm*, *D*, *R* and *Tp*), each module can automatically determine, by means of mathematical equations, the time stamp value $t_{stamp_{\mathsf{in}}}$ of the samples that should be consumed from each of its input ports, and the time stamp value $t_{stamp_{\mathsf{out}}}$ of the samples that should be produced to each of its output ports. In order to formulate these equations, we propose the generic model shown in Figure 4.3.

On the one hand, to determine the time stamp value of the samples that should be consumed from a TDF input port **n**, belonging to a TDF module **N**, we use the Equation 4.1. Where j_{N} is the number of times that the module **N** has been executed, Tm_{N} is the timestep associated to module **N**, Tp_{n} is the time step associated to port **n**, and k is an index going from 1 to the rate R_{n} associated to port **n**.

It is important to clarify that the number of times that a module has been executed (j) is increased when: the number of samples indicated by the rates associated to the input ports have been consumed, the processing() function of the module has been performed, and the number of samples indicated by the rates associated to the output ports have been produced.

On the other hand, to determine the time stamp value of the samples that should be produced by a TDF output port **m**, belonging to a TDF module **M**, we use the Equation 4.2. Where j_M is the number of times that the module **M** has been executed, Tm_M is the timestep associated to module **M**, D_m is the delay associated to port **m**, D_n is the delay associated to port **n** (TDF port connected to **m**), Tp_m is the time step associated to port **m**, and i is an index going from 1 to the rate R_m associated to the port **m**.

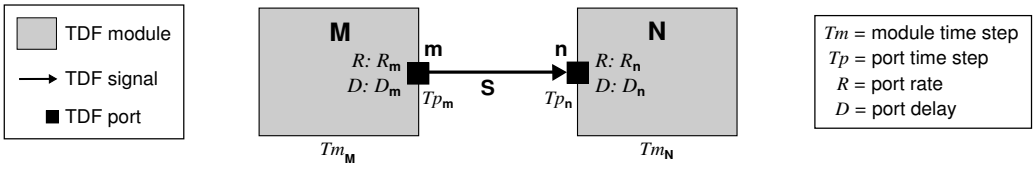

Figure 4.3: Example of a Generic TDF Cluster Composed by 2 TDF Modules and 1 TDF Signal.

$$t_{stamp_{in}} = (j_N * Tm_N) + ((k-1) * Tp_n) \qquad k = [1 \dots R_n] \qquad (4.1)$$

$$t_{stamp_{out}} = (j_M * Tm_M) + ((D_m + D_n) * Tp_m) + ((i-1) * Tp_m) \qquad i = [1 \dots R_m] \qquad (4.2)$$

Using the previously defined equations, the time stamps associated to the samples produced or consumed can be determined. For the example shown in Figure 4.1, the consumption and production of samples, together with their particular time stamps are illustrated in Figure 4.4.

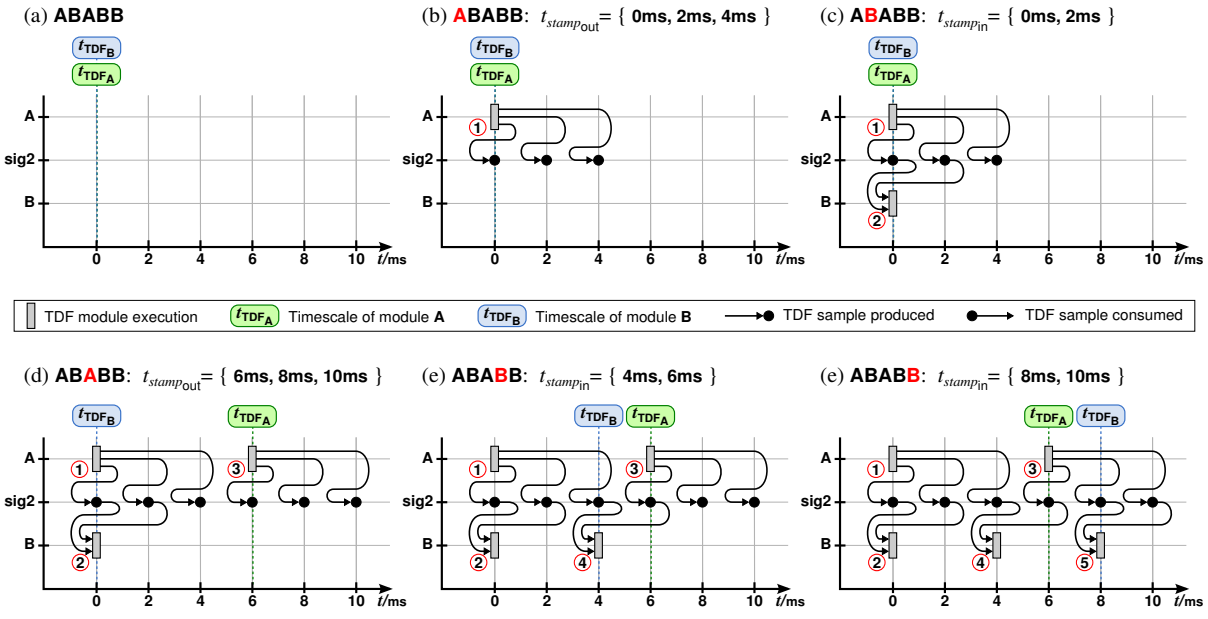

Figure 4.4: Time Management in TDF Ports Belonging to TDF Cluster shown in Figure 4.1.

4.2.2. Occurrence of Synchronization Issues

Having clarified how the time notion is handled inside TDF models, we can discuss the synchronization issues that can arise when these TDF models are interconnected, by means of TDF converter ports, with models described in the DE domain.

During a DE-TDF simulation, it is important to remember that the DE simulation time must remain monotonically increasing, and the actions generated from TDF clusters should not violate this principle. This means that for a TDF cluster, a valid static schedule must guarantee that the discrete events generated by a TDF cluster cannot happen earlier than the current DE time. These events can be named **synchronization actions**, which correspond with the *read operations* getting information from the DE domain, by means of input converter ports; and the *write operations* providing information to the DE domain, by means of output converter ports.

At present, the principle previously introduced is not guaranteed by the SystemC-AMS TDF MoC. As the schedule determined during elaboration only includes the order in which the TDF modules should be executed, regardless to their interactions with the DE domain, several causality problems may appear during the execution of this schedule. To illustrate the problem, we propose the model shown in Figure 4.5(a), which consists of two TDF modules **A** and **B**, respectively interconnected with two DE modules **X** and **Y**, through the DE signals **sig1** and **sig3**. Such signals are responsible for communicating the TDF and DE MoCs by means of the input and output converter ports, **A.in** and **B.out**, respectively.

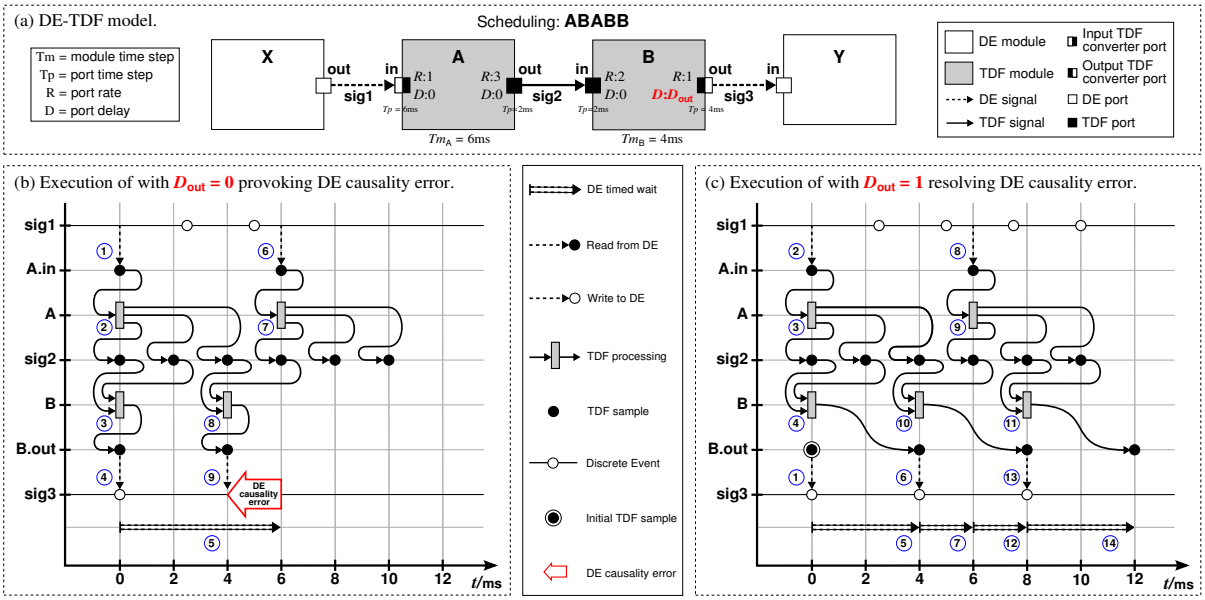

Figure 4.5: Transient Simulation of a TDF Cluster with DE-TDF Synchronization.

Considering the information provided in Section 4.2.1, we can represent the execution of the TDF model shown in Figure 4.5(a). Figure 4.5(b) shows the simulation trace when all the delay parameters are fixed to zero, and Figure 4.5(c) shows the simulation trace when the delay parameter of the output converter port **B.out** is fixed to one. These delay values were strategically selected to demonstrate that only including the execution order of TDF modules in the schedule, we cannot ensure the causality of a model in the DE domain. Causality also depends of the parameter values selected by the designer.

In the graphical representation that we developed, two kinds of time lines are represented: the ones having white circles denote a DE timescale, while the ones having black circles and grey boxes denote the TDF timescales. The position of a grey box indicates when a TDF module is activated and the solid arcs indicate its consumption and production of samples, in that order. For the sake of simplicity, the

trace details of the TDF ports are omitted. The dashed arcs denote either the sampling of a DE signal (*read synchronization operation*), or the generation of an event on a DE signal (*write synchronization operation*). The double dashed arrows indicate the advance of DE time by a timed wait() statement.

In Figure 4.5(b), following the schedule determined during elaboration, the simulation is executed until it detects a causality problem. When it occurs, the simulation is stopped and the designer has the responsibility to fix such problem. Details, about how the simulation is performed, are presented below.

① To activate the module **A** (first in the schedule), the DE signal **sig1** is sampled at the discrete event time $t_{DE} = 0$ ms. It makes a TDF sample available in the input converter port **A.in**, with a time stamp associated of 0 ms.

② This sample allows to activate the module **A** at $t_{TDF_A} = 0$ ms, which reads (consumes) the TDF sample available in the port **A.in** (with a time stamp associated of 0 ms); executes the **A** processing() function; and writes (produces) three TDF samples (with time stamps associated of 0 ms, 2 ms and 4 ms), through the output port **A.out**, to the TDF signal **sig2**.

③ Now, the module **B** (second in the schedule) can be activated at $t_{TDF_B} = 0$ ms, to consume two of the available samples (with time stamps associated of 0 ms and 2 ms) from the signal **sig2**, through the input port **B.in**; execute the **B** processing() function; and produce one sample (with time stamp associated of 0 ms) to the output converter port **B.out**. The time stamp associated to the samples stored in output converter ports indicates the DE time at which the write synchronization operations should be performed.

④ As the current DE time is $t_{DE} = 0$ ms, and the sample generated in the port **B.out** should be written in the DE domain at 0 ms, then, the write synchronization operation is performed on the DE signal **sig3**.

⑤ The next module to be executed is **A** (third in the schedule). As this module has a timestep $Tm_A = 6$ ms, its second execution should be performed at $t_{TDF_A} = 6$ ms. Besides, as this module has an input converter port **A.in**, a sample is required there, to start its execution. The generation of this sample in **A.in** indicates a read synchronization operation, which involves a DE time progression from 0 ms to 6 ms.

The DE time progression is scheduled, from the TDF cluster, by means of a wait() statement, which is registered in the DE simulation kernel. This operation suspends the execution of the TDF cluster until the DE time reaches the value provided as argument of the wait() statement. In the example, a wait(6ms) is registered.

⑥ When the execution is resumed, because the $t_{DE} = 6$ ms, the DE signal **sig1** is sampled. This makes a TDF sample available in the input converter port **A.in**, with a time stamp associated of 6 ms.

⑦ Now, module **A** is activated at $t_{TDF_A} = 6$ ms, to consume the TDF sample available in the port **A.in** (with a time stamp associated of 6 ms); execute the **A** processing() function; and produce

three TDF samples (with time stamps associated of 6 ms, 8 ms and 10 ms), through the output port **A.out**, to the TDF signal **sig2**. In this moment $t_{DE} = t_{TDF_A} = 6$ ms.

(8) Later, the module **B** (fourth in the schedule) can be activated at $t_{TDF_B} = 4$ ms, to consume two of the available samples (with time stamps associated of 4 ms and 6 ms) from the signal **sig2**, through the input port **B.in**; execute the **B** processing() function; and produce one sample (with time stamp associated of 4 ms) to the output converter port **B.out**.

(9) The time stamp associated to the sample stored in **B.out** indicates the DE time $t_{DE} = 4$ ms at which the write synchronization operation should be performed, but this constitutes a DE causality issue because the DE time cannot decrease. Remember that previously, for performing the last read synchronization operation (in (6)), the DE time was increased to $t_{DE} = 6$ ms.

To avoid the synchronization issue detected, thanks to the representation proposed, we can determine that the sample generated in the output converter port **B.out** at $t_{TDF_B} = 4$ ms, should be shifted 2 ms to be written in 6 ms (current DE time). This value $diff = 2$ ms, should be added in the port **B.out** to fix the causality in the model. It is possible increasing the delay attribute value associated to **B.out**. The delay value required D_{req_m} in a port **m** can be determined by the Equation 4.3.

$$D_{req_m} = \left\lceil \frac{diff}{Tp_m} \right\rceil \tag{4.3}$$

In Figure 4.5(c), we observe that adjusting the delay value in **B.out** and following the schedule determined during elaboration, the simulation is fully executed for a TDF cluster period $Tcls = 12$ ms. Details, about how this simulation is performed, are presented below.

(1) When a delay attribute $D = 1$ is assigned in **B.out**, an initial sample is available is such port when the simulation starts. This sample has a time stamp associated of 0 ms, indicating the time at which the first read synchronization operation should be performed. As initially $t_{DE} = 0$ ms, the sample is written on the **sig3** DE signal.

(2) As before, to activate the module **A** (first in the schedule), the DE signal **sig1** is sampled at the discrete event time $t_{DE} = 0$ ms. It makes a TDF sample available in the input converter port **A.in**, with a time stamp associated of 0 ms.

(3) This sample allows to activate the module **A** at $t_{TDF_A} = 0$ ms, which consumes the TDF sample available in the port **A.in** (with a time stamp associated of 0 ms); executes the **A** processing() function; and produces three TDF samples (with time stamps associated of 0 ms, 2 ms and 4 ms), through the output port **A.out**, to the TDF signal **sig2**.

(4) Now, the module **B** (second in the schedule) can be activated at $t_{TDF_B} = 0$ ms, to consume two of the available samples (with time stamps associated of 0 ms and 2 ms) from the signal **sig2**, through the input port **B.in**; execute the **B** processing() function; and produce one sample (with time stamp associated of 4 ms) to the output converter port **B.out**.

(5) As the current DE time is $t_{DE} = 0$ ms, and the sample generated in the port **B.out** should be written in the DE domain at 4 ms, then, a DE time progression should be scheduled from the TDF cluster, by means of a wait(4ms) statement.

(6) When the DE time reaches $t_{DE} = 4$ ms, the write synchronization operation is performed.

(7) The next module to be executed is **A** (third is the schedule). As this module has a timestep $Tm_A = 6$ ms, its second execution should be performed at $t_{TDF_A} = 6$ ms. Besides, as this module has an input converter port **A.in**, a sample is required there to start its execution. The generation of this sample in **A.in** indicates a read synchronization operation, which involves a DE time progression from 4 ms to 6 ms. This progression is scheduled from the TDF cluster, by means of a wait(2ms) statement.

(8) When the execution is resumed, because the $t_{DE} = 6$ ms, the DE signal **sig1** is sampled. This makes a TDF sample available in the input converter port **A.in**, with a time stamp associated of 6 ms.

(9) Now, module **A** is activated at $t_{TDF_A} = 6$ ms, to consume the TDF sample available in the port **A.in** (with a time stamp associated of 6 ms); execute the **A** processing() function; and produce three TDF samples (with time stamps associated of 6 ms, 8 ms and 10 ms), through the output port **A.out**, to the TDF signal **sig2**. In this moment $t_{DE} = t_{TDF_A} = 6$ ms.

(10) Later, the module **B** (fourth in the schedule) can be activated at $t_{TDF_B} = 4$ ms, to consume two of the available samples (with time stamps associated of 4 ms and 6 ms) from the signal **sig2**, through the input port **B.in**; execute the **B** processing() function; and produce one sample (with time stamp associated of 8 ms) to the output converter port **B.out**.

(11) As the module **B** has still samples to be consumed, it is activated at $t_{TDF_B} = 8$ ms, to consume the two available samples (with time stamps associated of 8 ms and 10 ms) from the signal **sig2**, through the input port **B.in**; execute the **B** processing() function; and produce one sample (with time stamp associated of 12 ms) to the output converter port **B.out**.

(12) As the current DE time is $t_{DE} = 6$ ms, and the second sample generated in the port **B.out** should be written in the DE domain at 8 ms, then, a DE time progression is scheduled from the TDF cluster, by means of a wait(2ms) statement.

(13) When the DE time reaches $t_{DE} = 8$ ms, the write synchronization operation is performed.

(14) Finally, as the current DE time is $t_{DE} = 8$ ms, and the third sample generated in the port **B.out** should be written in the DE domain at 12 ms, then, a DE time progression is scheduled from the TDF cluster, by means of a wait(4ms) statement.

When the DE time reaches $t_{DE} = 12$ ms, it is the end of the current TDF cluster period. Therefore, the sample available in the output converter port **B.out** represents the initial delay sample for the next TDF cluster period execution.

4.2.3. Preliminary Conclusions

- The discussed example shows how causality problems arise in multi-rate TDF models due to DE-TDF synchronization. At present, these problems can be only detected during simulation because the synchronization operations (read/write operations from/to DE) are not considered for determining the cluster's schedule during elaboration.
- The graphical representation used in Figures 4.5(b) and 4.5(c) is very helpful to understand the TDF and DE-TDF semantics, and diagnose any causality problem. This representation however reaches its limits when the DE-TDF model has a more complex topology, important rate differences, many delays, and feedback loops.
- We need another approach to represent the TDF models and their interactions with the DE time domain, determine the order in which the TDF modules' executions and their synchronization operations should be performed, and detect and analyze the causality problems present in the models. This approach is presented in Section 4.3.

4.3. CPN-Based Representation of DE and TDF Synchronization Interactions

The TDF MoC is based on the **Synchronous Data Flow (SDF)** formalism [35], [61], which considers models as a network (graph) of synchronous data flow blocks, as shown in Figure 4.6. This network is composed of a set of blocks (*nodes* interconnected by means of *directed arcs*), representing functions that are invoked (*fired*) to consume a known number of inputs (*input rate*) and produce a known number of outputs (*output rate*). The only condition required to fire each block is that the number of inputs required to be consumed, is available on each of the input arcs associated to the same block.

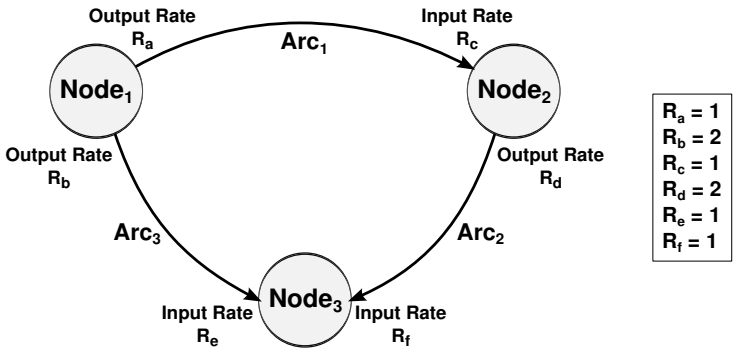

Figure 4.6: Example of a Synchronous Data Flow Graph (adapted from [35]).

This network of SDF blocks may also be represented by means of **Petri Nets (PN)** [62], as shown in Figure 4.7. This representation is defined as a directed bipartite graph, which interconnects *transitions* and *places* by means of a set of *directed arcs*. **Transitions** can be considered as functions invoked to consume a fixed number of inputs, and produce a fixed number of outputs; and **places** as the containers where such inputs and outputs are stored. These inputs and outputs are called **tokens**. In the case of PN, for *firing* a transition (execute the function that it represents), it should be *enabled*; this means that the number of inputs to be consumed (input required tokens) is available into each of the input places associated to the same transition.

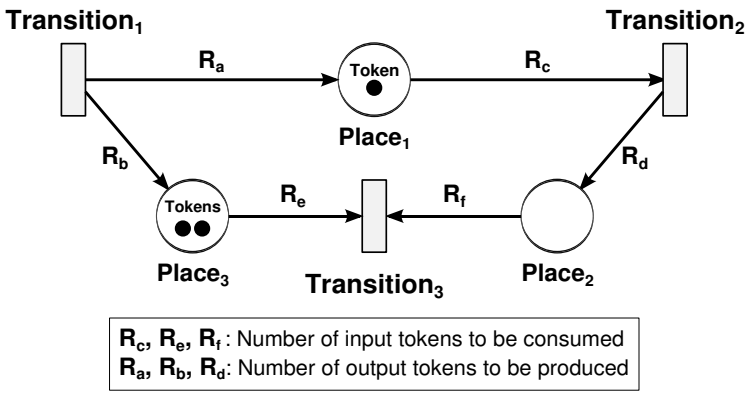

Figure 4.7: Example of a Petri Net.

Petri Nets are convenient to represent the TDF models, as shown in Figure 4.8. TDF modules are represented as *transitions*, TDF signals (channels) as *places*, TDF ports as *directed arcs*, TDF samples as *tokens*, TDF input rates as the *number of tokens to be consumed* by a transition, TDF output rates as the *number of tokens to be produced* by a transition, and TDF delays as the *initial tokens* that are stored in the input or output places associated to the transitions.

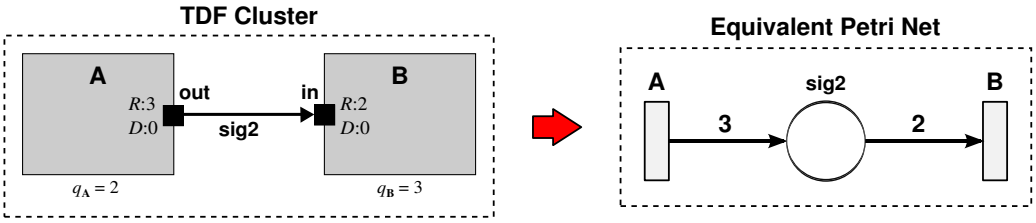

Figure 4.8: Equivalent Petri Net for a Basic TDF Cluster.

Using this representation, pre-simulations of TDF models, which does not include DE-TDF interactions, can be performed regardless to the time notions handled by TDF modules and classical TDF ports. Pre-simulations are performed following the execution rules of Petri Nets, and the ones imposed by TDF:

- In PN, a transition **T** can be fired when it is enabled.
- In PN, a transition **T** is enabled when the input required tokens are available in the input places associated to the same transition.
- In TDF, a module **M** (represented as a PN transition **T**) should be executed (fired) q_M times per period.
- In TDF, a module **M** (represented as a PN transition **T**) is immediately executed (fired) when it has enough samples in their input ports to be consumed, according to the input rate values associated.

As an example, we apply the PN and TDF rules to the example of Figure 4.8. Results are presented in Figure 4.9.

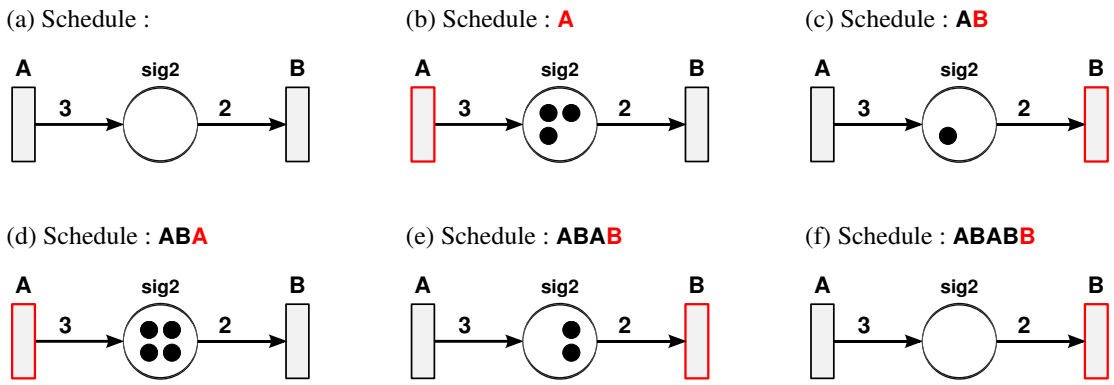

Figure 4.9: Execution of the Equivalent Petri Net shown in Figure 4.8.

These pre-simulations guarantee a static schedule, and bounded channels' memory for each TDF cluster:

- **Static Schedule**: the order in which the transitions (representing TDF modules) are fired for a TDF cluster period. This period is detected during the PN execution, when the PN reaches its initial state; it is when the number of tokens contained into each of its places is equal to the number of tokens initially contained there (before starting the execution).
- **Bounded Channels' Memory**: the maximum number of tokens contained into each place during the execution (for a TDF cluster period).

Considering that the execution order of TDF modules (which does not include interactions with the DE domain) can be found without involving the internal TDF time stamps, we can deduce that the time management inside a TDF cluster can be omitted in a representation proposed to analyze the synchronization issues between the DE kernel and the TDF MoC.

At present, we need to extend the representation of a TDF cluster as a Petri Net, which includes the timing information handled by the TDF converter ports in the model. This timing information corresponds to the DE timescale handled by the SystemC DE kernel. To this end, we have analyzed different PN extensions allowing the introduction of timing information, and selected the *Coloured Petri Nets* extensions.

4.3.1. Coloured Petri Nets (CPN) Extension

Coloured Petri Nets (CPN) is a discrete-event modeling language combining the capabilities of Petri nets (graphical representation for modeling concurrency, communication, and synchronization) with the capabilities of a high-level programming language (primitives for the definition of data types, for the description of data manipulation, and for the creation of compact and parametric models). This formalism allows to investigate different scenarios, to explore the system behavior, and to debug the system design. All these features and the CPN formal definitions presented below have been defined by Jensen and Kristensen in their book "Coloured Petri Nets, Modelling and Validation of Concurrent Systems" [17].

A CPN model, as shown in Figure 4.10, is a graphical representation, which contains *places*, *transitions*, *directed arcs*, *coloured tokens*, and *textual inscriptions*.

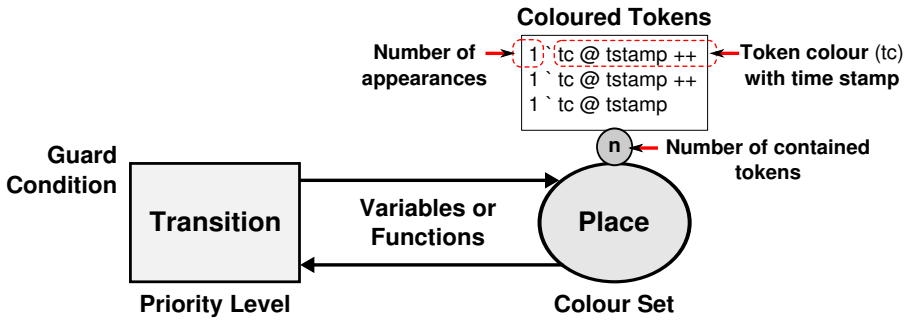

Figure 4.10: Example of a Coloured Petri Net Model.

- **Transitions**: represent the actions that can be performed in a model. They can associate both *priority levels* and *guard conditions*, which control the activation and execution of the represented actions.

- **Places**: represent the state of a model. This state is defined as the combination of the number of tokens contained in the places, and the data value attached to each token (*token colour*). In a CPN, the state can be only modified by the transitions' firing (execution of the actions allowed in the model).

- **Directed Arcs**: are the means by which the transitions and places are interconnected. They can associate *variables* or *functions*, which determine how the state of a model changes after each transition execution.

- **Coloured Tokens**: are tokens which have an attached data value, called *token colour*. This data value is defined by means of a type, i.e. integer, string, bool, etc. In CPN, each token respects a multiset notation consisting of a back-quote operator " ` ", which takes an integer as left argument specifying the number of appearances of the data value provided as right argument. When several tokens are grouped in a place, they are separated using the operator "++".

  In addition to the data value, a token can carry a second value called *time stamp*, useful for involving timing information of a model. This time stamp is added to the token using the operator "@", and indicates the time at which the token is ready to be consumed by an occurring transition.

- **Textual Inscriptions**: are expressions written in the CPN ML programming language [63] that can be attached to transitions, places or arcs.

  - **Transition Inscriptions**: can represent the *priority level* for the execution of a transition, by means of an integer value; or the *guard condition* limiting its execution.
  - **Place Inscriptions**: represent the type of tokens (*colour set*) contained in a place.
  - **Arc Inscriptions**: represent the expressions evaluated during simulation for consuming or producing tokens. These inscriptions can be *variables* or *functions*, which sometimes include timing information.

Note: when a CPN model contains timing information, it is called **Timed CPN** [64].

4.3.2. Representation of DE-TDF Models as Equivalent Timed CPN

In this section we develop an equivalent representation of TDF clusters and their interactions with the DE domain using *timed CPN*. This representation facilitates the understanding of the TDF simulation, the detection of timing inconsistencies, and the proposition of solutions for the synchronization issues presented in Section 4.2.2.

a. Equivalent CPN for TDF Modules

The first step for the construction of an equivalent timed CPN is the representation of TDF modules. To illustrate this representation, we consider the generic TDF module shown in Figure 4.11, where **M** is the module name, j_M is the number of times that the module **M** has been executed, and q_M is the number of times that the module **M** should be executed within a TDF period. As this TDF model is isolated from DE, there is no need of representing explicitly timing information.

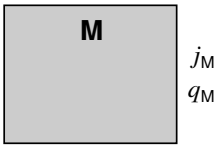

Figure 4.11: TDF Module to be Represented as an Equivalent CPN.

For representing this TDF module, we propose the equivalent CPN model shown in Figure 4.12: it is defined by means of one transition, one place, two directed arcs, and a set of equations written using the CPN ML language.

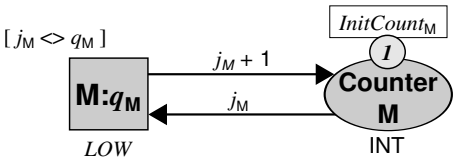

Figure 4.12: Equivalent CPN for the TDF Module shown in Figure 4.11.

$$\textbf{colset}\ \mathsf{INT} = int \tag{4.4}$$

$$\textbf{var}\ j_M : \mathsf{INT} \tag{4.5}$$

$$\textbf{val}\ LOW = 3 \tag{4.6}$$

$$\textbf{val}\ InitCount_M = 1`0 \tag{4.7}$$

Initially, the TDF module execution action is represented by means of a transition with three defined textual inscriptions:

- A name **M:q_M**, to identify the transition.
- A guard [j_M<>q_M], which represents a Boolean expression used to evaluate whether the transition is enabled.
- A priority level *LOW*, defined in Equation 4.6, which restricts the transition occurrence.

In addition, the TDF module's current execution number j_M is stored in a place with three defined textual inscriptions:

- A name **Counter M**, which identifies the place.
- A color set INT, defined in Equation 4.4, which indicates the token data type that can be contained therein. In this case the place can contain only integer values.
- An initial marking $InitCount_M$, defined in Equation 4.7, which specifies the initial tokens of the place. The initial marking (*1`0*) indicates that one token with an integer data value equal to zero is contained in the place, before starting the CPN execution.

Later, to complete the TDF module representation, directed arcs link the defined transition and the defined place. Textual inscriptions next to arcs indicate that the j_M value is incremented when the **M:q_M** transition is fired.

Finally, in order to simplify the notation used for representing the TDF modules, we propose the reduced CPN shown in Figure 4.13. It hides the internal functionality of a TDF module.

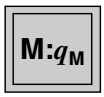

Figure 4.13: Reduced CPN for the TDF Module shown in Figure 4.11.

b. Equivalent CPN for TDF Connections

The second step for the construction of an equivalent timed CPN is the representation of TDF connections. To illustrate this representation, we consider the generic TDF model shown in Figure 4.14, where **M** is the name of the source module, **N** is the name of the sink module, **m** is the name of the output port belonging to **M** module, **n** is the name of the input port belonging to **N** module, **R** is the rate attribute associated to a port, **D** is the delay attribute associated to a port, and **S** is the name of the signal connecting both **M** and **N** modules. As this TDF model is isolated from DE, there is no need of representing explicitly timing information.

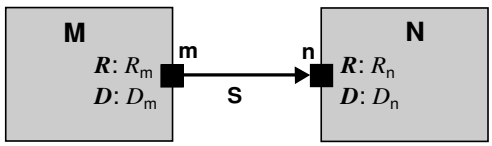

Figure 4.14: TDF Connections to be Represented as an Equivalent CPN.

For representing the TDF connections involved in the model, we propose the equivalent CPN model shown in Figure 4.15: it is defined by means of two transitions, one place, two directed arcs, and a set of equations written using the CPN ML language.

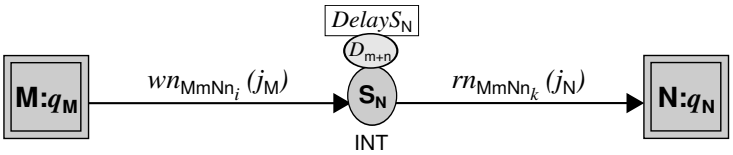

Figure 4.15: Equivalent CPN for the TDF Connections shown in Figure 4.14.

$$\textbf{fun}\ wn_{\mathsf{MmNn}_i}(j_\mathsf{M} : \mathsf{INT}) = (j_\mathsf{M} * R_\mathsf{m}) + (D_\mathsf{m} + D_\mathsf{n}) + i \qquad i = [1 \ldots R_\mathsf{m}] \tag{4.8}$$

$$\textbf{fun}\ rn_{\mathsf{MmNn}_k}(j_\mathsf{N} : \mathsf{INT}) = (j_\mathsf{N} * R_\mathsf{n}) + k \qquad k = [1 \ldots R_\mathsf{n}] \tag{4.9}$$

$$\textbf{val}\ DelayS_\mathsf{N} = 1\text{`}1 +\!+ 1\text{`}2 +\!+ \ldots +\!+ 1\text{`}D_{\mathsf{m+n}} \qquad D_{\mathsf{m+n}} = D_\mathsf{m} + D_\mathsf{n} \tag{4.10}$$

Initially, the TDF modules **M** and **N** are represented using the reduced model previously presented in Figure 4.13, and the TDF signal **S** is represented using a place with three textual inscriptions:

- The name $\mathbf{S_N}$ (signal **S** connected to an input port of module **N**).
- The color set defined in Equation 4.4, which indicates that only tokens with integer data values can be contained in the place.
- The initial marking defined in Equation 4.10, indicating the multiset of delay tokens associated to the TDF signal. Note that the delay tokens number of the place $D_{\mathbf{m+n}}$ is the addition of the delay attributes associated to the interconnected ports.

Later, the transition representing the producer (source) module $\mathbf{M}{:}\boldsymbol{q}_\mathbf{M}$ is linked to the $\mathbf{S_N}$ place, using a directed arc annotated with the Equation 4.8. This equation calculates the *identifier* of the token that should be produced when $\mathbf{M}{:}\boldsymbol{q}_\mathbf{M}$ is fired. The *identifier* represents the position of the token in the $\mathbf{S_N}$ place.

Similarly, the $\mathbf{S_N}$ place is linked to the transition representing the consumer (sink) module $\mathbf{N}{:}\boldsymbol{q}_\mathbf{N}$, using a directed arc annotated with the Equation 4.9. This equation calculates the identifier of the token that should be consumed when $\mathbf{N}{:}\boldsymbol{q}_\mathbf{N}$ is fired.

Note that in multi-rate models, the number of arcs linking transitions and places, are determined by the involved port rates (in the equations, it is represented using the i and k index).

Finally, in order to simplify the notation used for representing the TDF connections, we propose the reduced CPN shown in Figure 4.16. It replaces the textual inscription associated to each directed arc, by the rate attribute values associated to each TDF port.

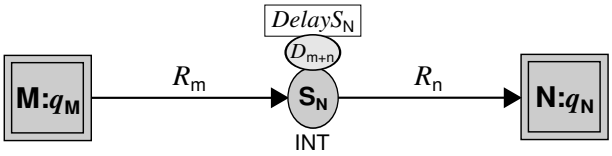

Figure 4.16: Reduced CPN for the TDF Connections shown in Figure 4.14.

c. Equivalent Timed CPN for TDF Input and Output Converter Ports

The third step for the construction of an equivalent CPN model is the representation of the input and output converter ports. This representation adds the timing information required to synchronize the read and write synchronization operations among the DE and DT domains.

On the one hand, to illustrate the representation of input converter ports, we consider the generic TDF model shown in Figure 4.17, where **M** is the name of the module, **m** is the name of the input converter port, **S** is the name of the DE signal connected to **m** port, R_m is the rate attribute associated to **m** port, D_m is the delay attribute associated to **m** port, Tm_M is the timestep associated to **M** module, and Tp_m is the timestep associated to **m** port.

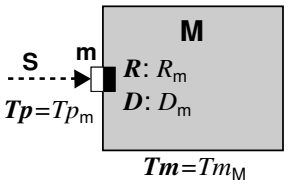

Figure 4.17: TDF Input Converter Port to be Represented as an Equivalent CPN.

For the TDF input converter port shown in Figure 4.17, we introduce the equivalent CPN model shown in Figure 4.18: it is defined by means of several transitions, places, directed arcs, and equations (4.12 – 4.21) written using the CPN ML language.

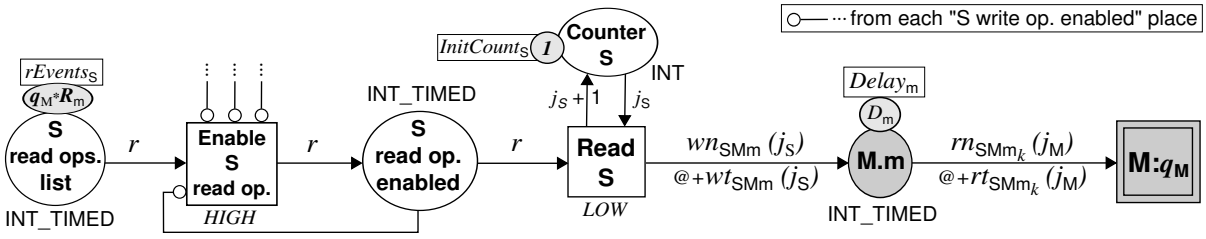

Figure 4.18: Equivalent CPN for the TDF Input Converter Port shown in Figure 4.17.

$$t_\mathsf{CPN} : \textit{CPN simulation time} \tag{4.11}$$

$$\textbf{colset}\ \mathsf{INT\_TIMED} = \textit{int timed} \tag{4.12}$$

$$\textbf{var}\ r : \mathsf{INT\_TIMED} \tag{4.13}$$

$$\textbf{fun}\ wn_\mathsf{SMm}\ (j_\mathsf{S} : \mathsf{INT}) = j_\mathsf{S} + D_\mathsf{m} + 1 \tag{4.14}$$

$$\textbf{fun}\ wt_\mathsf{SMm}\ (j_\mathsf{S} : \mathsf{INT}) = (j_\mathsf{S} * Tp_\mathsf{m}) + (D_\mathsf{m} * Tp_\mathsf{m}) - t_\mathsf{CPN} \tag{4.15}$$

$$\textbf{fun}\ rn_{\mathsf{SMm}_k}\ (j_\mathsf{M} : \mathsf{INT}) = (j_\mathsf{M} * R_\mathsf{m}) + k \qquad k = [1 \ldots R_\mathsf{m}] \tag{4.16}$$

$$\textbf{fun}\ rt_{\mathsf{SMm}_k}\ (j_\mathsf{M} : \mathsf{INT}) = (j_\mathsf{M} * Tm_\mathsf{M}) + ((k-1) * Tp_\mathsf{m}) - t_\mathsf{CPN} \qquad k = [1 \ldots R_\mathsf{m}] \tag{4.17}$$

$$\textbf{val}\ Delay_\mathsf{m} = 1`1@0 + +1`2@(1 * Tp_\mathsf{m}) + + \ldots + +1`D_\mathsf{m}@(D_\mathsf{m} - 1) * Tp_\mathsf{m} \tag{4.18}$$

$$\textbf{val}\ rEvents_\mathsf{S} = 1`1@0 + +1`2@(1 * Tp_\mathsf{m}) + + \ldots + +1`x@(x-1) * Tp_\mathsf{m} \qquad x = [1 \ldots (q_\mathsf{M} * R_\mathsf{m})] \tag{4.19}$$

$$\textbf{val}\ InitCount_\mathsf{S} = 1`0 \tag{4.20}$$

$$\textbf{val}\ HIGH = 1 \tag{4.21}$$

- The **S read ops. list** place stores the reading synchronization events defined in Equation 4.19 for one cluster period.
- The **Enable S read op.** transition enables a read synchronization operation at time t_{CPN}.
- The **S read op. enabled** place stores the read synchronization operation that is enabled at time t_{CPN}.
- The **Counter S** place stores the number of read synchronization operations that have been executed at t_{CPN}.
- The **Read S** transition represents the read synchronization operation to be performed from the DE to the DT domain.
- The **M.m** place stores the available tokens, which can be consumed by the **M:q_{M}** transition. This place represents the input converter port **m**, belonging to the module **M**. Note that the initial marking of this place (Equation 4.18) is present when **m** has a delay attribute associated.

On the other hand, to illustrate the representation of output converter ports, we consider the generic TDF model shown in Figure 4.19, where **M** is the name of the module, **m** is the name of the output converter port, **S** is the name of the DE signal connected to **m** port, R_{m} is the rate attribute associated to **m** port, D_{m} is the delay attribute associated to **m** port, Tm_{M} is the timestep associated to **M** module, and Tp_{m} is the timestep associated to **m** port.

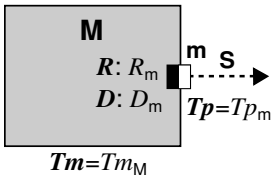

Figure 4.19: TDF Output Converter Port to be Represented as an Equivalent CPN.

For the TDF output converter port shown in Figure 4.19, we introduce the equivalent CPN model shown in Figure 4.20: it is also defined by means of several transitions, places, directed arcs, and equations (4.23 – 4.28) written using the CPN ML language.

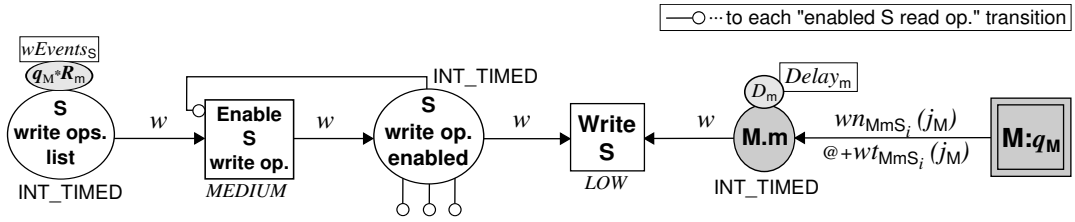

Figure 4.20: Equivalent CPN for the TDF Output Converter Port shown in Figure 4.19.

$$t_{\mathrm{CPN}} : \mathit{CPN\ simulation\ time} \tag{4.22}$$

$$\textbf{var}\ w : \mathsf{INT\_TIMED} \tag{4.23}$$

$$\textbf{fun}\ wn_{\mathrm{MmS}_i}(j_{\mathrm{M}} : \mathsf{INT}) = (j_{\mathrm{M}} * R_{\mathrm{m}}) + D_{\mathrm{m}} + i \qquad i = [1 \ldots R_{\mathrm{m}}] \tag{4.24}$$

$$\textbf{fun}\ wt_{\mathrm{MmS}_i}(j_{\mathrm{M}} : \mathsf{INT}) = (j_{\mathrm{M}} * Tm_{\mathrm{M}}) + (D_{\mathrm{m}} * Tp_{\mathrm{m}}) + ((i-1) * Tp_{\mathrm{m}}) - t_{\mathrm{CPN}} \qquad i = [1 \ldots R_{\mathrm{m}}] \tag{4.25}$$

$$\textbf{val}\ Delay_{\mathrm{m}} = 1`1@0 + + 1`2@(1 * Tp_{\mathrm{m}}) + + \ldots + + 1`D_{\mathrm{m}}@(D_{\mathrm{m}} - 1) * Tp_{\mathrm{m}} \tag{4.26}$$

$$\textbf{val}\ wEvents_{\mathrm{S}} = 1`1@0 + + 1`2@(1 * Tp_{\mathrm{m}}) + + \ldots + + 1`x@(x-1) * Tp_{\mathrm{m}} \qquad x = [1 \ldots (q_{\mathrm{M}} * R_{\mathrm{m}})] \tag{4.27}$$

$$\textbf{val}\ MEDIUM = 2 \tag{4.28}$$

- The **S write ops. list** place stores the writing synchronization events defined in Equation 4.27 for one cluster period.
- The **Enable S write op.** transition enables a write synchronization operation at time t_{CPN}.
- The **S write op. enabled** place stores the write synchronization operation enabled at time t_{CPN}.
- The **M.m** place stores the available tokens, which should be written in the DE domain by the **M:**$q_{\mathbf{M}}$ transition. This place directly represents the output converter port **m**, belonging to module **M**. Note that the initial marking of this place (Equation 4.26) is present when **m** has a delay attribute associated.
- The **Write S** transition represents the DE write synchronization operation to be performed from the DT to DE domain.

In the models shown in Figures 4.18 and 4.20, a new color set INT_TIMED (defined in Equation 4.12) is associated to some of the defined places: it indicates that the tokens stored in the places contain not only an identifier (integer value), but also a time stamp indicating when they can be consumed; this time stamp is added to the token using the operator "@".

Besides, three transition priority levels (*HIGH*, *MEDIUM* and *LOW*) are defined in Equations 4.21, 4.28, and 4.6. The *HIGH* priority transitions are reserved to enable the DE read synchronization operations, the *MEDIUM* ones to enable the DE write synchronization operations, and the *LOW* ones to enable the TDF executions.

Note that the arc inscriptions defined in Equations 4.14 – 4.17 and Equations 4.24 – 4.25 calculate the identifier and the time stamp of the tokens that should be consumed and produced at each CPN simulation time t_{CPN}. The formulation of all equations, used as arc inscriptions in the model, has been derived following the Equations 4.1 and 4.2, previously defined in Section 4.2.1.

Later, to complete the representation, we use a particular type of arc defined in PN. It is the *inhibitor arc*, which can be connected from a place **P** to a transition **T**, as shown in Figure 4.21. It establishes the precondition that the transition **T** may only be enabled and fired when the place **P** is empty.

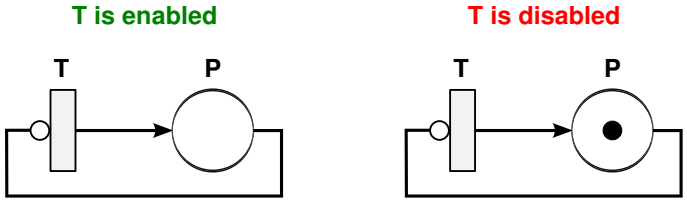

Figure 4.21: Example of a Petri Net's Inhibitor Arc.

We define some inhibitor arcs in our equivalent CPN model for controlling the execution of the transitions, which enable the read and write synchronization operations for a time t_{CPN}:

- The execution of each **Enable S read op.** transition is inhibited by the **S read op. enabled** place directly linked to it, and by each **S write op. enabled** place present in the model, as shown in Figure 4.18.
- The execution of each **Enable S write op.** transition is inhibited by the **S write op. enabled** place directly linked to it, as shown in Figure 4.20.

Finally, in order to simplify the notation used for representing the TDF converter ports, we propose the reduced timed CPN shown in Figures 4.22 and 4.23. They hide the textual inscriptions associated to each directed arc.

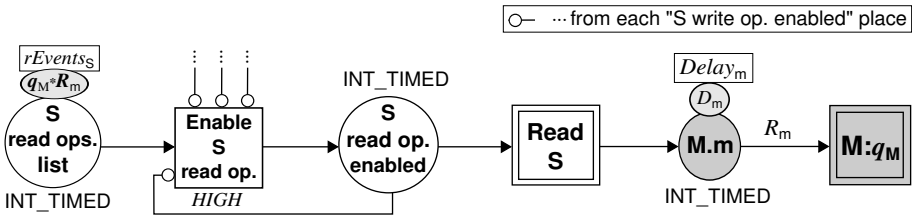

Figure 4.22: Reduced CPN for the TDF Input Converter Port shown in Figure 4.17.

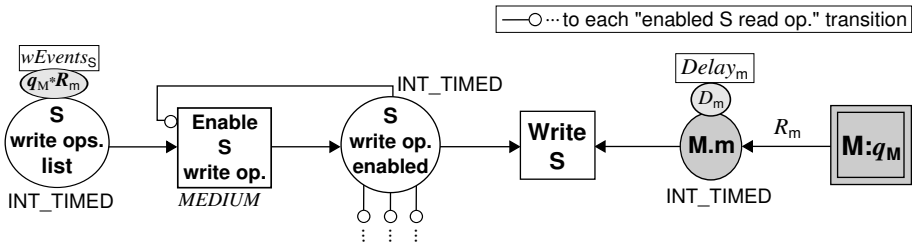

Figure 4.23: Reduced CPN for the TDF Output Converter Port shown in Figure 4.19.

d. Integration of Equivalent CPN Models

By combining the reduced transformation rules defined in Figures 4.13, 4.16, 4.22 and 4.23, any TDF cluster can be represented using a timed CPN. As an example, Figure 4.24 shows the equivalent timed CPN model for the DE-TDF model proposed in Figure 4.5(a).

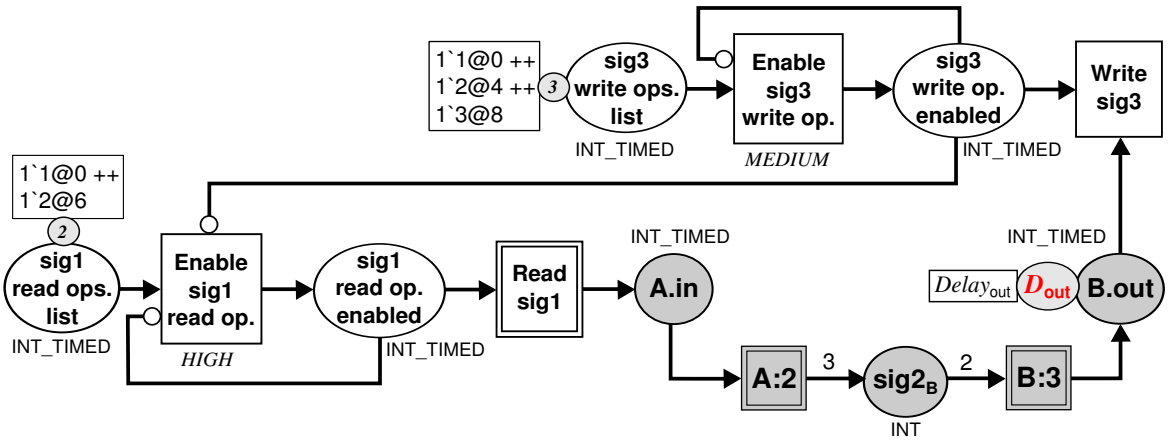

Figure 4.24: Equivalent CPN the DE-TDF Model shown in Figure 4.5(a).

The proposed representation was validated using CPN Tools [65], a tool for editing and simulating CPN. For the example shown in Figure 4.24, we verified that the semantics of the CPN equivalent model is properly represented; its execution when $D_{\text{out}} = 1$ yields the simulation trace shown in Figure 4.5(c), for a TDF cluster period; and its execution when $D_{\text{out}} = 0$ yields the simulation trace shown in Figure 4.5(b), which is interrupted due to DE-TDF causality problems.

4.3.3. Preliminary Conclusion

In this section, we have proposed equivalent CPN models that can be built for representing a TDF cluster and its interactions with the DE domain. This representation may be executed once before simulation, for analyzing the computability of the TDF cluster. A TDF cluster will be computable when its execution for a cluster period is not interrupted due to DE-TDF causality problems.

Based on the CPN execution rules and the restrictions imposed for executing TDF models, we have developed a method for analyzing equivalent CPN models, detecting the causality problems and fixing them. This method is presented in Section 4.4.

4.4. DE-TDF Pre-Simulation Analysis

In this section, we introduce a DE-TDF pre-simulation analysis method, which determines the computability of a TDF cluster, represented by means of an equivalent CPN model. This analysis method is based on the principle that a TDF cluster is computable, when the execution of its equivalent CPN model is performed without interruptions for a cluster period `T_cpn`. Following this principle, our method is defined by means of three phases:

- **Phase 1 – Transitions Firing**: all the enabled transitions of an equivalent CPN model are fired. During this phase:
  - The transitions are fired according to their priority levels.
  - The `schedule` required for planning the TDF cluster execution is constructed under certain conditions (only low priority transitions are added in the `schedule`).
  - The CPN execution time `t_cpn` is increased, without exceeding the `T_cpn` value.

  When no more transitions are enabled, the phase 2 begins.

- **Phase 2 – Final State Verification**: the CPN is evaluated for determining if its final state is reached. During this phase, we have three possible scenarios:
  - *The final state is reached the first time that this phase is executed*: it means that DE-TDF causality problems are not present in the model. In this scenario, the model is identified *computable* and the analysis method ends.
  - *The final state is not reached*: it means that causality problems are present in the model, and they should be solved. In this scenario, the model is identified *non-computable* and the phase 3 begins.
  - *The final state is reached, but previously the model was identified non-computable*: it means that all model's causality problems were successfully identified. In this scenario, the changes proposed for solving such causality problems are notified to the designer, and the analysis method ends.

- **Phase 3 – Unlocking and correction**: the objects (transitions and places), which disable the execution of the CPN for a time `t_cpn` are identified; and the attributes associated to these objects are temporarily modified. Once the modifications are performed, new CPN transitions are then enabled, thus the phase 1 is re-executed.

Using our analysis method, when the TDF model is computable, we can determine the schedule required for planning the TDF cluster execution; otherwise, we can identify and notify to the designer, a solution for the causality problems present in the TDF cluster.

In order to summarize the analysis method previously described, we propose the algorithm shown in Listing 4.1. It takes as arguments: the cpn structure on which the analysis is performed; the schedule structure where the execution order is stored; and the T_cpn cluster period, which corresponds to the period of the represented TDF cluster.

```
1  bool analyze_computability (cpn, schedule, T_cpn) {
2
3     bool computable = true;
4     bool final_state = false;
5     double t_cpn = 0;
6
7     fire_enabled_transitions (cpn, schedule, computable, t_cpn, T_cpn);
8     final_state = verify_final_cpn_state (cpn, computable);
9     while (!final_state) {
10       computable = false;
11       unlock_cpn_and_determine_delay_changes (cpn, t_cpn);
12       fire_enabled_transitions (cpn, schedule, computable, t_cpn, T_cpn);
13       final_state = verify_final_cpn_state (cpn, computable);
14    }
15
16    if (!computable)
17       print_required_delay_changes (cpn);
18
19    return computable;
20 };
```

Listing 4.1: Algorithm to analyze the computability of a TDF cluster by means of an equivalent CPN model.

In this algorithm, initially the model is assumed computable, the CPN final state is assumed not reached, and the CPN execution time t_cpn is initialized at zero (Listing 4.1, lines 3–5).

The first called function (Listing 4.1, line 7) implements the first phase of our method. It is responsible for firing all enabled transitions and adding to the schedule, the order and the times at which the low priority transitions are fired. The timescale handled during the CPN execution corresponds to the DE timescale handled by the represented DE-TDF cluster. Details about how the transitions are fired in an equivalent CPN model, are presented in Section 4.4.1.

Once the CPN is locked (transitions are disabled), it is necessary to check if the CPN final state is reached (second phase of our method). It is implemented by means of the function shown in Listing 4.1, line 8. Details about how the final state is verified in an equivalent CPN model, are presented in Section 4.4.2.

If the final state is directly reached, the algorithm returns true indicating that the schedule was completed and that no causality problems were found. Otherwise, the model is marked as non-computable (Listing 4.1, line 10), the CPN is temporarily unlocked, and the delay changes required to solve the causality problems are determined (Listing 4.1, line 11). It corresponds to the third phase of our method. Details about how the causality problems are detected and fixed in an equivalent CPN model, are presented in Section 4.4.3 and Section 4.4.4.

Once the CPN is temporarily unlocked, the execution continues until a new locked scenario is found (Listing 4.1, line 12). This means that the CPN analysis is performed while it has not yet reached its final state. Finally, when the CPN final state is reached and the model is marked as non-computable, the delay changes required to solve the causality problems are presented (Listing 4.1, lines 16–17).

Using an implementation of this algorithm in C++, equivalent CPN models can be analyzed. For example, the model show in Figure 4.24 can be analyzed in two scenarios ($D_{\text{out}} = 0$ and $D_{\text{out}} = 1$). Results of these analysis are summarized in Table 4.1.

Initial Delays	**Schedule**	**Computability**	**Proposed Changes**
$D_{\text{out}} = 0$	0 ms – **read sig1, A, B, write sig3**	*false*	$D_{\text{out}} = 1$
$D_{\text{out}} = 1$	0 ms – **write sig3, read sig1, A, B** 4 ms – **write sig3** 6 ms – **read sig1, A, B, B** 8 ms – **write sig3**	*true*	none

Table 4.1: Analysis Results of the CPN Model shown in Figure 4.24.

When $D_{\text{out}} = 0$, the causality problems are detected, the schedule is not valid, and the delay changes are proposed. When $D_{\text{out}} = 1$, the CPN directly reaches its final state and the schedule is constructed, indicating the execution order and the DE times at which the TDF modules and their interactions with the DE domain should be performed.

More details about the implementation and execution of this analysis are presented in the next sections. We introduce how the transitions are fired, how the CPN final state is verified, and how the causality problems are detected and fixed in an equivalent CPN model.

4.4.1. Firing Transitions in Equivalent CPN Models

The enabled transitions are fired in an equivalent CPN model according to the defined priority levels and the timed CPN execution rules [64]. In our approach, the *HIGH* priority transitions are reserved to enable the DE read operations, the *MEDIUM* ones to enable the DE write operations, and the *LOW* ones to enable the TDF executions. Once a transition is fired, it can be added to the schedule according to the next rules:

- Only the low priority transitions are added in the schedule, because they represent the executions of TDF modules (**M:q_{M}** transitions), and the interactions of such TDF modules with the DE domain (**Read S** and **Write S** transitions).
- A transition is added to the schedule while the model is considered computable.

In order to summarize the method for firing transitions, we introduce the algorithm shown in Listing 4.2. It takes as arguments: the cpn structure on which the analysis is performed; the schedule structure where the execution order is stored; the model computability status computable, the current CPN execution time t_cpn, and the T_cpn cluster period. In this algorithm, initially the CPN is assumed enabled to be executed at time t_cpn, and the minimum CPN time used for increasing t_cpn during execution, is initialized at zero (Listing 4.2, lines 3–4).

```
1  void fire_enabled_transitions (cpn, schedule, computable, t_cpn, T_cpn) {
2
3    bool disabled_cpn = false;
4    double tmin = 0;
5
6    while (!disabled_cpn) {
7      fire_enabled_high_priority_transitions (cpn);
8      fire_enabled_medium_priority_transitions (cpn);
9
10     if (!computable)
11       fire_enabled_low_priority_transitions (cpn);
12     else
13       fire_and_push_enabled_low_priority_transitions (cpn, schedule);
14
15     disabled_cpn = true;
16     tmin = search_minimum_tstamp (cpn);
17     if ((tmin > t) && (tmin != T_cpn)) {
18       t_cpn = tmin;
19       disabled_cpn = false;
20     }
21   }
22 };
```

Listing 4.2: Algorithm to fire the CPN enabled transitions.

For a time t_cpn, following the CPN execution rules, transitions are fired according to the defined priority levels (Listing 4.2, lines 7–13). If the model is computable, the low priority transitions are added to the schedule (Listing 4.2, line 13). Once the model is non-computable, the construction of the schedule halts: the low priority transitions are fired, but they are not added to the schedule (Listing 4.2, line 11).

The example shown in Figures 4.25 and 4.26, illustrates how the transitions are fired in the equivalent CPN model shown in Figure 4.24 (with $D_{\text{out}} = 0$), according the CPN execution rules and the priority levels associated to each transition (for a CPN execution time $t_{\text{CPN}} = 0\,\text{ms}$).

(a) The first transition enabled (**Enable sig1 read op.**) has a *HIGH* priority level associated, and has a least one available token (with $t_{stamp} = 0\,\text{ms}$) to be consumed (at $t_{\text{CPN}} = 0\,\text{ms}$). When this transition is fired, it consumes the token "1@0" from the **sig 1 read ops. list** place, and produces the token "1@0" to the **sig1 read op. enabled** place. This action indicates that the first read synchronization operation will be enabled to be performed in the TDF cluster at time $t_{\text{DE}} = 0\,\text{ms}$.

(b) The second transition enabled (**Enable sig3 write op.**) has a *MEDIUM* priority level associated, and has a least one available token (with $t_{stamp} = 0\,\text{ms}$) to be consumed (at $t_{\text{CPN}} = 0\,\text{ms}$). When this transition is fired, it consumes the token "1@0" from the **sig 3 write ops. list** place, and produces the token "1@0" to the **sig3 write op. enabled** place. This action indicates that the first write synchronization operation will be enabled to be performed in the TDF cluster at time $t_{\text{DE}} = 0\,\text{ms}$.

(c) The third transition enabled (**Read sig1**) has a *LOW* priority level associated, and has one available token (with $t_{stamp} = 0\,\text{ms}$) to be consumed (at $t_{\text{CPN}} = 0\,\text{ms}$). When this transition is fired, it consumes the token "1@0" from the **sig 1 read op. enabled** place, and produces the token "1@0" to the **A.in** place. This action indicates that the first read synchronization operation will be performed in the TDF cluster at $t_{\text{DE}} = 0\,\text{ms}$, making a TDF sample available in the input converter port **A.in**, with a

time stamp associated of 0 ms.

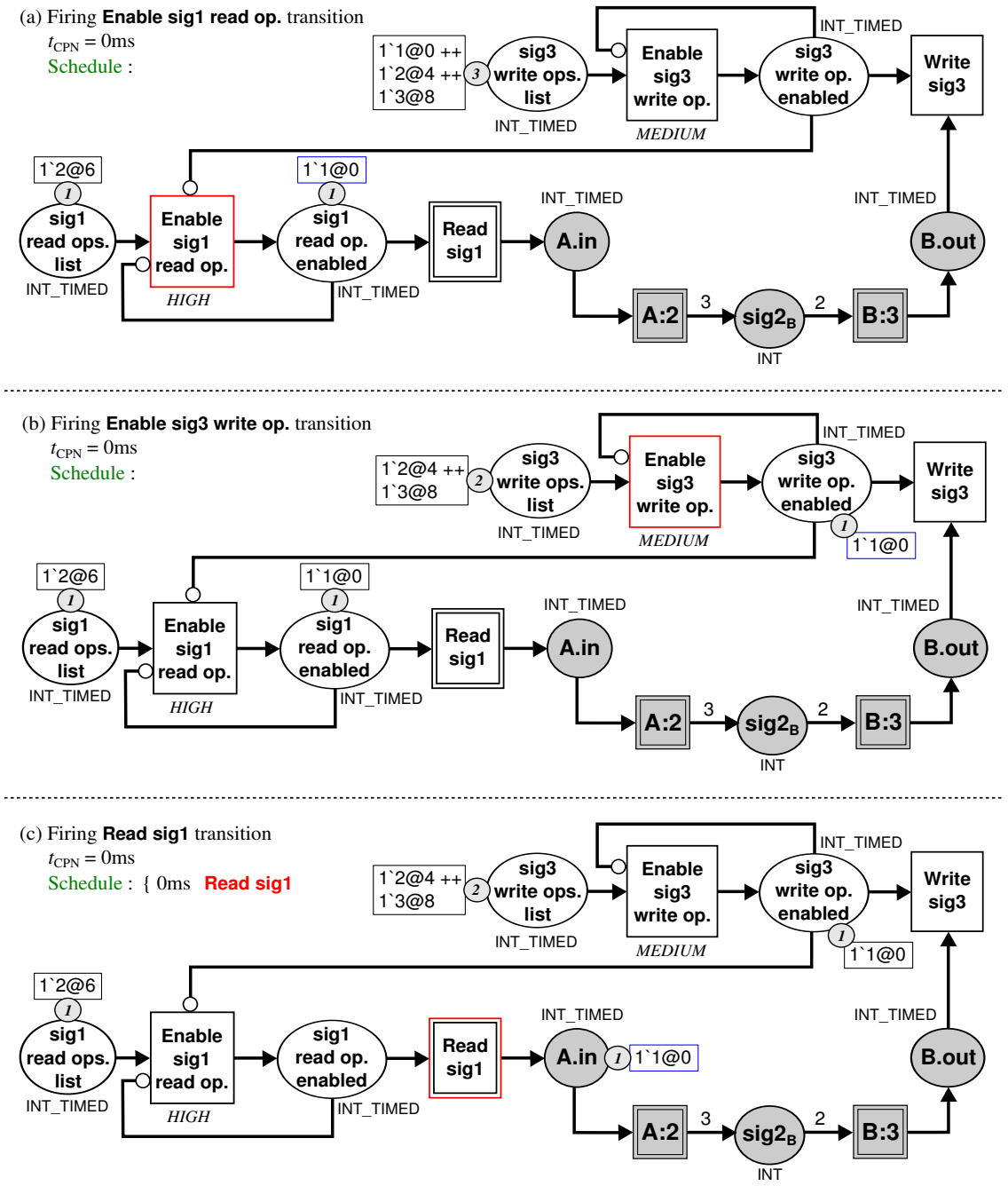

Figure 4.25: Firing Transitions in the Equivalent CPN Model shown in Figure 4.24 (I).

(d) The fourth transition enabled (**A:2**) has a *LOW* priority level associated, and has one available token (with $t_{stamp} = 0$ ms) to be consumed (at $t_{CPN} = 0$ ms). When this transition is fired, it consumes the token "1@0" from the **A.in** place, and produces three tokens to the **sig2$_B$** place. This action represents the first execution of module **A** in the TDF cluster, making three TDF samples available in the TDF signal **sig2**.

(e) The fifth transition enabled (**B:3**) has a *LOW* priority level associated, and has at least two available tokens to be consumed. When this transition is fired, it consumes two tokens from the **sig2$_B$** place, and produces one token (with $t_{stamp} = 0$ ms) to the **B.out** place. This action represents

the first execution of module **B** in the TDF cluster, making one TDF sample available in the output converter port **B.out**, with a time stamp associated of 0 ms.

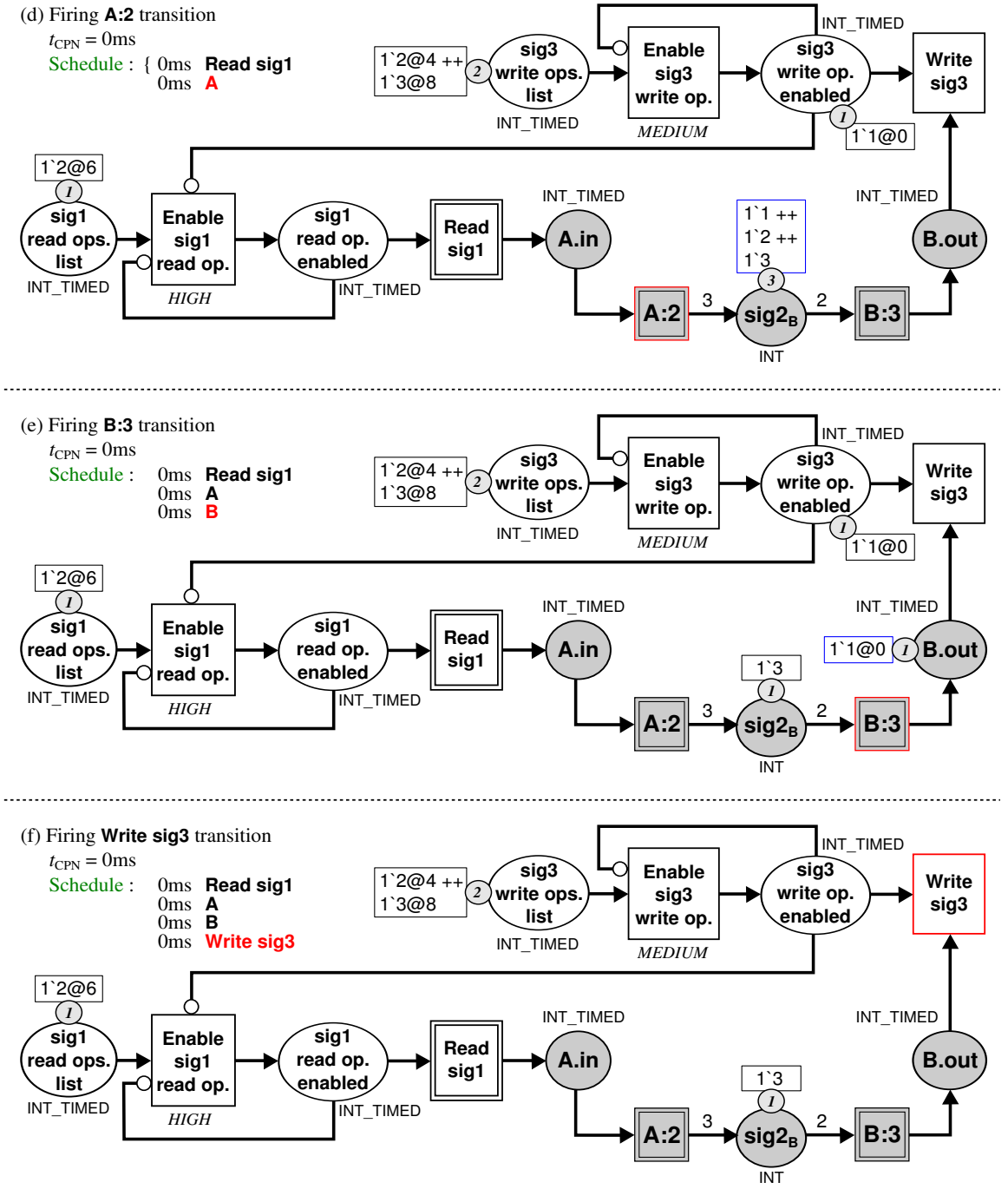

Figure 4.26: Firing Transitions in the Equivalent CPN Model shown in Figure 4.24 (II).

(f) The sixth transition enabled (**Write sig 3**) has a *LOW* priority level associated, and has into each of their input places, one available token (with t_{stamp} = 0 ms) to be consumed (at t_{CPN} = 0 ms). When this transition is fired, it consumes the tokens "1@0" from the **sig 3 write op. Enabled** and the **B.out** places. This action indicates that the first write synchronization operation will be performed in the TDF cluster at t_{DE} = 0 ms.

Note: observe that only the low priority transitions are added to the schedule.

To continue with the description of the algorithm shown in Listing 4.2, we present the condition for increasing the CPN execution time t_cpn in equivalent CPN models. As long as time t_cpn is different to the T_cpn period and no more transitions are enabled, the time t_cpn is increased to the minimum time stamp value contained in the CPN places (Listing 4.2, lines 15–19). As the time increase enables new transitions, the algorithm will be re-executed (Listing 4.2, lines 6–21). An example of this condition is shown in Figure 4.27.

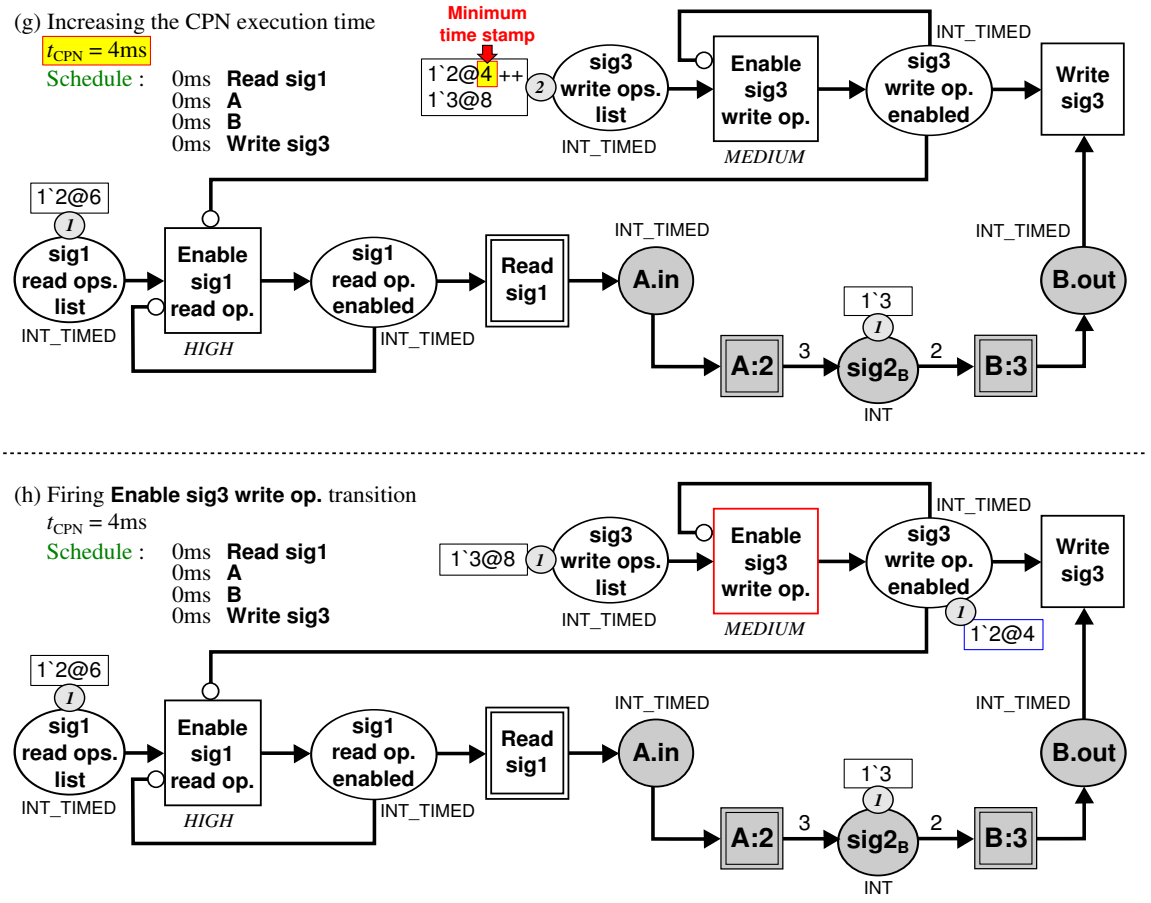

Figure 4.27: Increasing CPN Execution Time in the Equivalent CPN Model shown in Figure 4.26(f).

(g) When no more transitions are enabled in the equivalent CPN model at time $t_{CPN} = 0$ ms, the next transition to be enabled is the one, which has in their input places the token with the minimum time stamp (**Enable sig3 write op.** transition). This minimum value ($t_{stamp} = 4$ ms) represents the time to which t_{CPN} will be increased.

(h) When t_{CPN} is increased to 4 ms, the enabled transition is fired to consume the token "2@4" from the **sig 3 write ops. list** place, and produce the token "2@4" to the **sig3 write op. enabled** place. This action indicates that the second write synchronization operation will be enabled to be performed in the TDF cluster at time $t_{DE} = 4$ ms.

4.4.2. Verification of Final States in Equivalent CPN Models

Three conditions should be verified for ensuring the final state in equivalent CPN models:

- All the initial synchronization tokens have been consumed from the **S read ops. list** place and the **S write ops. list** place.
- The **M:q_M** transitions have been fired q_M times.
- The number of tokens contained in the **M:q_M** and **S_N** places is equal to the initial number of tokens contained there, when the execution began.

When these conditions are not satisfied, we can ensure that the equivalent CPN representation has not been completely executed for a TDF cluster period. An example of verification is shown in Figure 4.28:

- Synchronization tokens should yet be consumed from the **sig1 read ops. list** place and the **sig3 write ops. list** place.
- **A:2** transition has been executed once instead of twice; and **B:3** transition has been executed once instead three times.
- The number of tokens contained in the **$sig2_B$** place ($current_n = 1$) are greater than the number of tokens initially contained there ($initial_n = 0$).

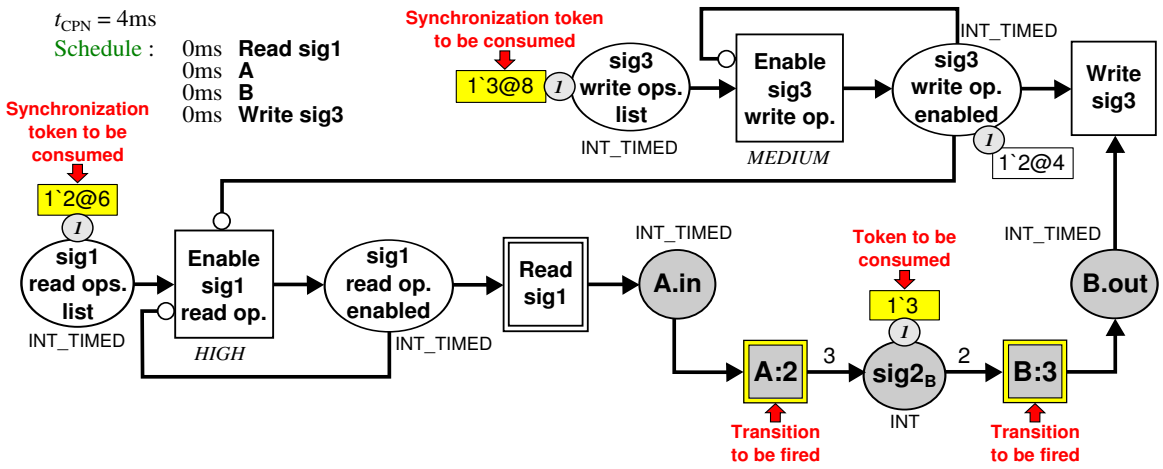

Figure 4.28: Verifying the Final State in the Equivalent CPN Model shown in Figure 4.27(h).

4.4.3. Detection of Synchronization Issues in Equivalent CPN Models

The causality problems in a TDF cluster occur when during the execution of its equivalent CPN representation, one or more of the following conditions are fulfilled: it is locked, it has not reached its final state, a DE write operation is required, and the sample to be written in the DE domain has not yet been generated by a TDF output converter port.

In the equivalent CPN model, the detection of this problem corresponds to identifying the locked **Write S** transition, because its **S write op. enabled** connected place has one token indicating that

the write operation should be realized at time t_{CPN}; and its **M.m** connected place has no token to be consumed at time t_{CPN}. An example of this detection is shown in the yellow block of Figure 4.29:

- The CPN is locked and has not reached its final state.
- The place **sig3 write op. enabled** indicates that a write synchronization operation should be performed at time $t_{CPN} = 4\,ms$.
- The place **B.out** has no token to be consumed at time $t_{CPN} = 4\,ms$.
- The **Write sig3** transition is locked, then the write synchronization operation cannot be performed.

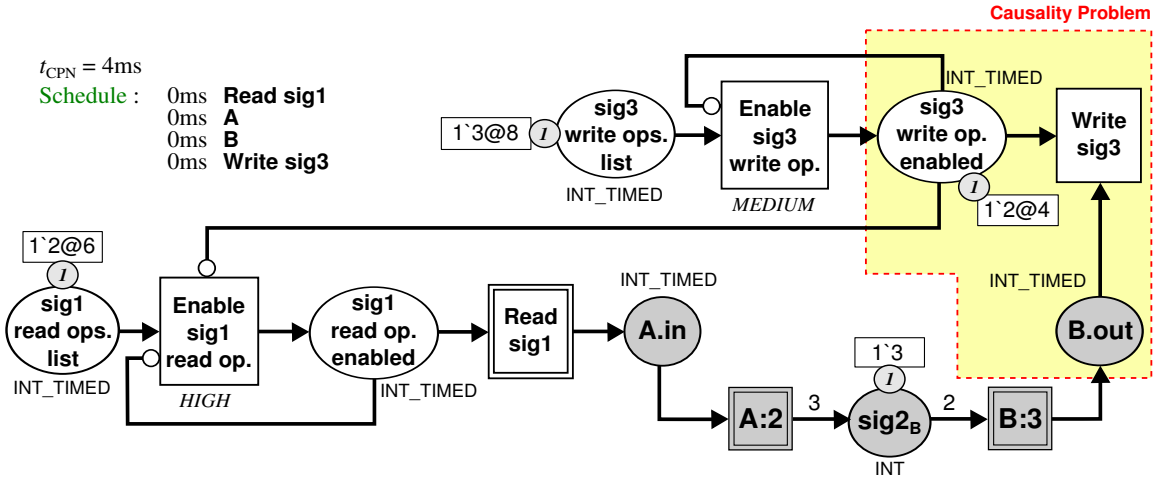

Figure 4.29: Detecting Synchronization Issues in the Equivalent CPN Model shown in Figure 4.28.

4.4.4. Fixing Synchronization Issues in Equivalent CPN Models

Once a synchronization issue has been detected in an equivalent CPN model, the delay changes required to solve such issue are determined. It consists in selecting the locked **Write S** transition (source of the causality issue), deleting the token contained in its **S write op. enabled** connected place, and increasing the delay attribute associated to the **M.m** connected place. This delay attribute is increased by the number of samples contained in the **S write op. enabled** connected place. After these modifications, the result is a CPN able to continue its execution. For the example shown in Figure 4.29:

- The token "2@4" is deleted from the to the **sig3 write op. enabled** place.
- The delay attribute in the **B.out** place is increased ($D_{out} = 0 \rightarrow D_{out} = 1$).

4.4.5. Preliminary Conclusions

In this section, we have proposed a method for analyzing, before simulation, the computability of a TDF cluster represented by means of an equivalent CPN model. Using this method:

- When a TDF cluster is computable, we should automatically determine the static schedule to be used during simulation, including TDF module executions and DE-TDF interactions.

- When a TDF cluster is not computable, we should detect all the causality problems presented in a model for a TDF cluster period, and propose solutions to fix them, by means of delay attributes' modifications.

Unlike SystemC-AMS, causality problems in TDF models are not detected one by one during simulation. Then, the designer does not need to perform several complete simulations to determine all the delay attribute changes required in the model.

4.5. Conclusion and Outlook

In this chapter, after analyzing the TDF MoC semantics, we demonstrated that the causality problems arising in multi-rate TDF clusters interacting with the DE domain can be detected and resolved before simulation.

We also showed that our approach of analyzing an equivalent CPN constructed from a TDF cluster for these problems yields a valid schedule for causal TDF clusters. In addition to the order of the TDF module activations and their interactions with the DE domain, this schedule also includes the DE times at which they should be performed.

On the one hand, the approach can be used to support the synchronization between the DE and TDF MoCs. It allows the construction of TDF clusters by means of equivalent CPN models, and the analysis of such equivalent models for a TDF cluster period. This analysis, in the case of computable clusters, will allow the TDF cluster scheduling, ensuring that the simulation will not be stopped by temporal inconsistencies. In the case of non-computable clusters, it will avoid the execution of simulations that cannot be finished due to temporal inconsistencies with the DE domain.

On the other hand, the approach cannot be used to support the synchronization between DE and other domains, without imposing the TDF semantics on them, which means that all the TDF models have to follow the time constraints imposed by the TDF MoC. In some cases, forcing a model to follow the TDF semantics may affect the simulation accuracy. For this reason, we believe that the DE-TDF synchronization approach cannot be the only one considered for synchronizing several domains inside the same multi-disciplinary simulation environment.

In order to define a new method for handling the synchronization in a multi-disciplinary simulation environment, in Chapter 5 we introduce a hierarchical synchronization approach, which is based on the principle that two different MoCs may be synchronized if, and only if, at least one synchronization method is defined to handle the different timescales involved between them. In this way, for example, the synchronization between the DE and TDF MoCs will follow the approach presented in this chapter; but the synchronization among the DE and other MoCs requires the definition of new specific synchronization methods.

Based on this hierarchical synchronization approach, we will define a multi-disciplinary simulator prototype called SystemC MDVP.

CHAPTER

5 SystemC Multi-Disciplinary Virtual Prototyping (MDVP) Simulator Prototype

Contents

5.1. Introduction

In this chapter, we introduce the modeling, synchronization, generic elaboration and simulation principles used to define a simulator prototype called SystemC Multi-Disciplinary Virtual Prototyping (MDVP), which is implemented as an extension of the SystemC design modeling language. It is a prototype designed to support the modeling and simulation of heterogeneous systems, by means of well-separated Models of Computation (MoCs).

Principles presented in this chapter are the result of multiple discussions carried out by a working group of the Laboratory of Computer Sciences of Paris 6, within the framework of the European project CATRENE CA701 Heterogeneous Inception (H-INCEPTION) [66].

In Section 5.2, we introduce the definition of *Model of Computation* in SystemC MDVP.

In Section 5.3, based on the block-oriented approach followed by SystemC and SystemC AMS, we present the modeling principles used to describe models in SystemC MDVP. We define the elements, which can be interconnected to represent particular behaviors under different MoCs; the means by which these elements are related; how the computation and communication are handled and well-separated; and how the hierarchical modeling is allowed.

In Section 5.4, we introduce the definition of *solver* in SystemC MDVP. We describe the synchronization principle introduced to ensure that the interactions between different MoCs are not limited by Discrete Time (DT) semantics, to allow the definition of generic elaboration and simulation methods, and to simplify the addition of MoCs. We clarify how the heterogeneity is handled, and how the MoCs to be included in SystemC MDVP can be related to each other following a hierarchical approach.

In Section 5.5, we introduce the hierarchical elaboration and simulation principles proposed to prepare and execute multi-disciplinary models in SystemC MDVP. Based on the elaboration and simulation phases implemented by the SystemC Discrete Event (DE) simulation kernel, we present an extension of these phases that can be performed on models regardless of the MoCs involved.

In Section 5.6, we present an overview about the implementation of the SystemC MDVP simulation kernel. We describe the classes created to represent the simulation objects, the building methods associated to these objects, and the abstract methods allowing the elaboration and simulation phases in the simulator prototype. In addition, we explain how the SystemC DE and the SystemC MDVP simulation kernels are interconnected.

In Section 5.7, we introduce a methodology to add models of computation to the SystemC MDVP simulation kernel. These MoCs can be implemented at different hierarchical levels to ensure interactions with one or more of the already defined models of computation.

Finally, in Section 5.8, we conclude this chapter discussing the MDVP simulation approach.

5.2. Model of Computation in SystemC MDVP

Model of Computation (MoC) is the term used to define the *time abstraction*, *computation*, *communication*, *synchronization*, *elaboration* and *simulation semantics* under which the components of a model can be described.

- The **time abstraction** is the representation of time handled by the components of a model (e.g. continuous, discrete, sampled).

- The **computation semantics** defines how a model is processed. In SystemC MDVP, the MoC computation semantics is implemented by means of *modules*.

- The **communication semantics** defines how the information is transmitted between the components of a model. In SystemC MDVP, the MoC communication semantics is implemented by means of *ports*, *interfaces* and *channels*.

- The **synchronization semantics** defines how the components of a model can interact with other ones described in different MoCs. In SystemC MDVP, the MoC synchronization semantics is implemented by means of *solvers*.

- The **elaboration and simulation semantics** defines how the components of a model are analyzed, initialized and prepared for the model execution. In SystemC MDVP, the MoC elaboration and simulation semantics is also implemented by means of *solvers*.

In SystemC MDVP, *modules*, *ports*, *interfaces* and *channels* are the components used by the designer to describe a particular behavior, as introduced in Section 5.3; and *solvers* are the objects automatically instantiated by the simulator to handle the interactions, elaboration and simulation of MoCs, as introduced in Section 5.4.

5.3. Modeling in SystemC MDVP

5.3.1. Model Components

SystemC MDVP follows the block oriented approach of SystemC, presented in Section 2.2.1, where a system can be represented by the composition and connection of different components: *modules*, *ports* implementing *interfaces*, and *channels*.

In SystemC MDVP, as shown in Figure 5.1, *modules* belonging to different MoCs contain *ports*, that are connected to *channels* through *interfaces*. These *ports*, *channels* and *interfaces* also belong to particular MoCs.

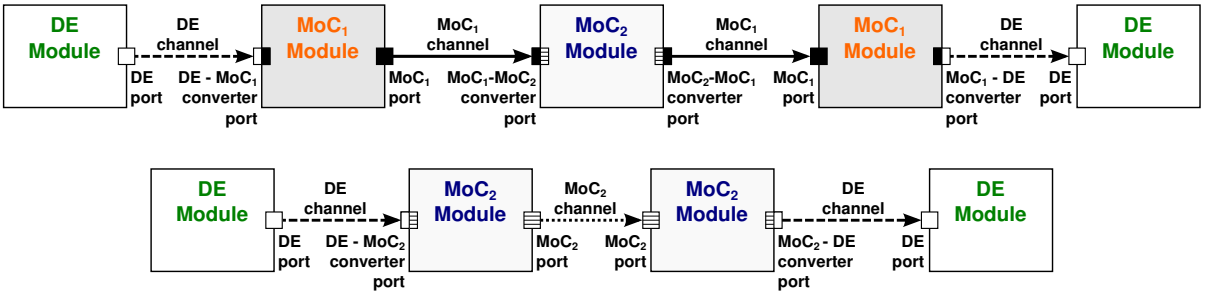

Figure 5.1: SystemC MDVP Components.

- **Modules**: are the objects which process the information, and encapsulate the behaviors associated to a particular MoC. They are identified with a unique name in the model, and have a set of *ports* through which they communicate the information that they are responsible for processing. According to the definition of the MoC to which a module belongs, it can be implemented by means of a sequential function, or can be predefined as a primitive ready to be instantiated by the designer.

- **Ports**: are the objects through which the modules communicate with other modules belonging, or not, to the same MoC in which they are defined. This means that, despite being defined in a particular MoC, ports can ensure not only the internal MoC communication, but also the *communication* and *data synchronization* between different MoCs. They are divided in:

  - **Classical ports**: are the objects through which two modules, belonging to the same MoC in which the ports are defined, can communicate. For the example shown in Figure 5.1, a pair of classical ports, belonging to a **MoC_2**, is used to relate two modules, belonging to the same **MoC_2**, by means of a channel also belonging to the **MoC_2**.

  - **Converter ports**: are the objects through which two modules, belonging to different MoCs can communicate. As shown in Figure 5.1, this communication can be performed in the input or output of a module. We can classify this type of ports in:

    * **Input converter ports**: which perform the communication from a module belonging to a **MoC_1**, to a module belonging to a **MoC_2** (MoC in which the port is defined), by means of a channel belonging to a **MoC_1**.

    * **Output converter ports**: which perform the communication from a module belonging to a **MoC_2** (MoC where the port is defined), to a module belonging to a **MoC_1**, by means of a channel belonging to a **MoC_1**.

- **Interfaces and Channels**: *interfaces* define the set of methods to access the *channels*, which are the data structures containing the information transmitted between modules. As channels are associated to particular MoCs in a model, they can be connected between ports following the rules presented below:

  - A channel, belonging to a **MoC_1**, can be connected between classical ports belonging to the same **MoC_1**.

  - A channel, belonging to a **MoC_1**, can be connected from a classical port belonging to a **MoC_1**, to an input converter port belonging to a **MoC_2**. In this case, the input converter port ensures the *data synchronization* from the **MoC_1** to the **MoC_2**.

  - A channel, belonging to a **MoC_1**, can be connected from an output converter port belonging to a **MoC_2**, to a classical port belonging to a **MoC_1**. In this case, the output converter port ensures the *data synchronization* from the **MoC_2** to the **MoC_1**.

Thanks to the last described components, the *computation* and *communication* are well-separated in a model: regardless of the MoCs included, computation is handled by means of modules; and communication by means of ports implementing interfaces, and channels.

5.3.2. MoC Hierarchical Approach

Using SystemC MDVP, designers have the task of implementing the modules, belonging to one or several MoCs, and linking them using predefined ports and channels. This task should be accomplished following a *MoC hierarchical approach*, which allows the simulator to automatically encapsulate, into structures called *clusters*, the modules interconnected and described in the same MoCs. The creation of these clusters will facilitate the *synchronization*, *elaboration* and *simulation* of multi-disciplinary models.

Our approach is based on the principle that a set of modules described in a single model of computation **MoC$_2$**, and interconnected using signals belonging to the same **MoC$_2$**, can interact with other sets of modules, through converter ports belonging to the **MoC$_2$**, if the two following conditions are satisfied:

- The other sets of modules are described in one, and only one, model of computation **MoC$_1$**.
- There are converter ports, defined in **MoC$_2$**, which ensure the *data synchronization* between the **MoC$_1$** and the **MoC$_2$**.

To illustrate the principle, we consider the model shown in Figure 5.2, where each set of interconnected modules, belonging to the same MoC, interacts with modules described in only one different MoC. In this figure, we explicitly represent the set of interconnected modules, with the aim of highlighting that each one of them is delimited by converter ports (relating only two MoCs by set).

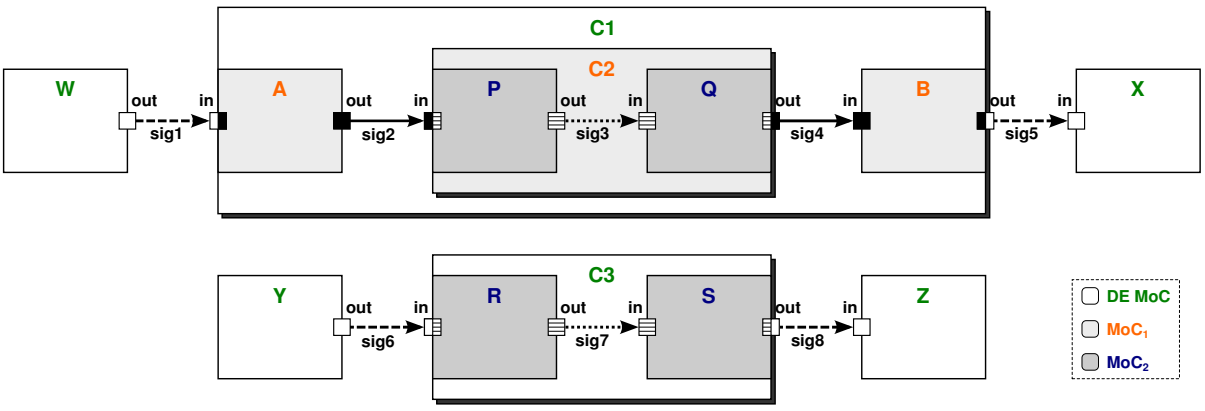

Figure 5.2: Example of Identification of Clusters in a SystemC MDVP Model.

Once the model is defined by the designer, the simulator encapsulates the modules as shown in Figure 5.3. Using this representation, we can observe that:

- A model in SystemC MDVP is hierarchically organized according to the models of computation involved.
- Clusters are considered as black boxes, which behave as the modules located in the same hierarchical level in which they are defined.

- Clusters can contain modules and other clusters.
- Clusters are always limited by converter ports defined to perform interactions between two particular MoCs: the MoC which handles the hierarchical level where the cluster is located, and the one which handles the hierarchical level where the cluster's components are located.

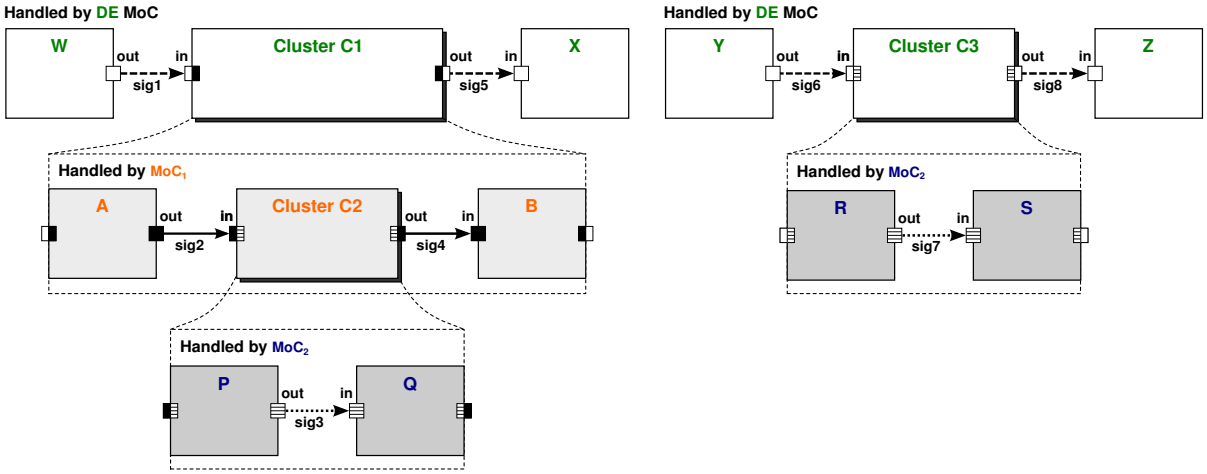

Figure 5.3: Encapsulation of SystemC MDVP Modules into Clusters, for the Example shown in Figure 5.2.

Note: As the SystemC MDVP simulator kernel is implemented on top of the SystemC DE kernel, we consider that the highest hierarchical level will be always handled by the DE MoC.

5.4. Solver in SystemC MDVP

An advantage of the *MoC hierarchical approach* introduced in Section 5.3.2, is that the interactions between the MoCs associated to the components instantiated in a model, can be easily identified and handled by means of particular elements called *solvers*.

A **solver** in SystemC MDVP is the element defined by the *MoC designer* (inside a model of computation), which will be automatically instantiated by the simulator in a particular cluster for:

- Handling the *time synchronization* between a pair of *master-slave* MoCs. The *master* is the MoC which will impose the synchronization constraints to be followed by the cluster components, and the *slave* is the MoC in which the solver is defined.
- Handling the *elaboration* and *simulation* of the components encapsulated in the cluster, in which this solver is instantiated.

Note: In SystemC MDVP, multiple solvers can be defined in a same MoC.

Details about how the *time synchronization* is handled in a MoC hierarchy, are presented in Section 5.4.1; and details about how the *elaboration* and *simulation* are generically handled, are presented in Section 5.4.2.

5.4.1. MoC Synchronization

As previously introduced in Section 2.3.1, the current implementation of the SystemC AMS language standard defines only one direct interaction method between the DE MoC and the Timed Data Flow (TDF) MoC. In consequence, the other MoCs included in such implementation are actually executed under the control of the TDF MoC.

In order to address this drawback, we introduce for SystemC MDVP, a new architectural model clarifying the interaction methods to be implemented between different MoCs. In this model, as shown in Figure 5.4, a hierarchical organization of MoCs is considered, where the **DE MoC** (1 in Figure 5.4) is the base for establishing the *time synchronization constraints* to be respected by other MoCs located in lower hierarchical levels.

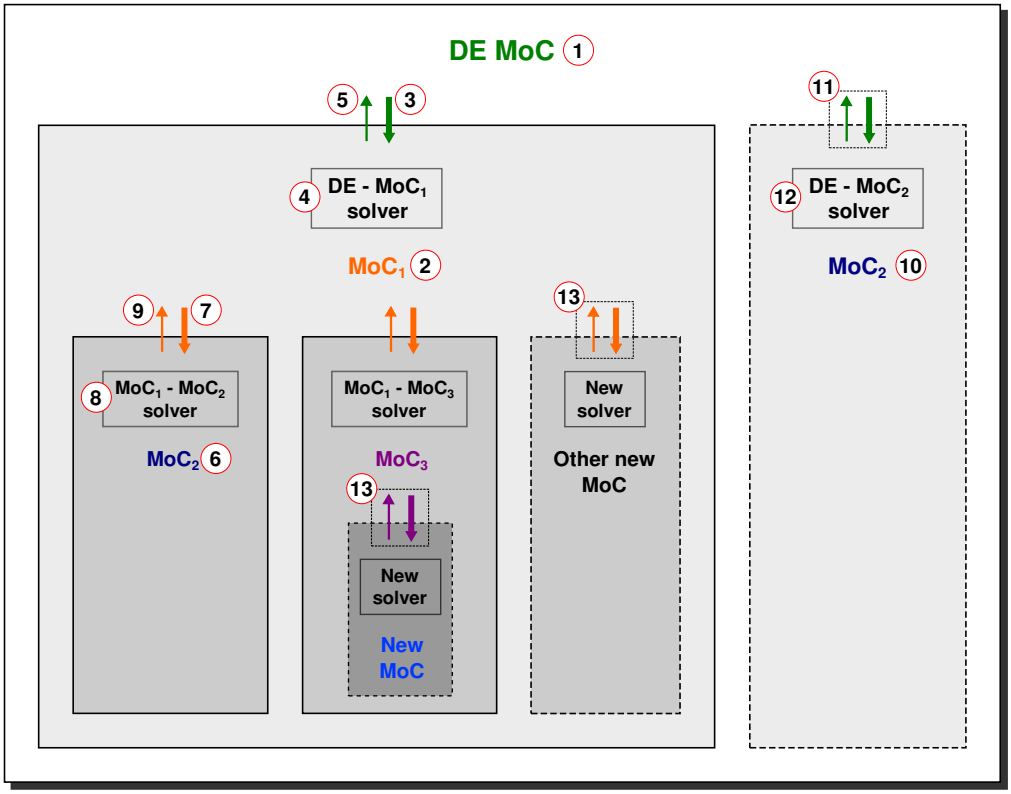

Figure 5.4: SystemC MDVP Architectural Model.

In this approach, interactions between the **DE MoC** and the **MoC$_1$** (2 in Figure 5.4) are performed by a **DE-MoC$_1$ solver** (4 in Figure 5.4):

- During *elaboration*, instances of the **DE-MoC$_1$ solver** are responsible for the analysis and preparation of **MoC$_1$** clusters, which want to interact with the **DE MoC**.
- During *simulation*, the same solver instances are responsible for the **DE-MoC$_1$** *time synchronization*, achieved in three phases:
  - First, the DE kernel imposes the *time synchronization constraints* for the **MoC$_1$** cluster executions (3 in Figure 5.4).

  - Second, the **DE-MoC$_1$ solver** instances proceed with the execution of the elements contained inside the **MoC$_1$** clusters.
  - Third, when the **DE-MoC$_1$ solver** instances reach *synchronization actions*, they return the control to the DE kernel (⑤ in Figure 5.4), via wait() statements to request a reactivation in the future.

An example of a MoC, which can be located under DE, is the TDF MoC. In this case, the interactions between the DE and TDF MoCs could be performed by a DE-TDF solver, which during *elaboration* can execute the mechanism formalized in Chapter 4, for analyzing a TDF cluster, detecting its synchronization issues, and determining the schedule and synchronization actions for a cluster period. During *simulation*, first, the DE kernel can provide the current simulation time; second, the DE-TDF solver can follow the schedule previously determined to perform the execution of the modules and their interactions with the DE MoC; and third, when a DE-TDF synchronization operation is required, the solver can return the simulation control to the DE kernel.

Following the same approach, interactions between the **MoC$_1$** and the **MoC$_2$** (⑥ in Figure 5.4) will be performed by a **MoC$_1$-MoC$_2$ solver** (⑧ in Figure 5.4):

- During *elaboration*, instances of the **MoC$_1$-MoC$_2$ solver** are responsible for the analysis and preparation of **MoC$_2$** clusters, which want to interact with the **MoC$_1$**.
- During *simulation*, the same solver instances handle the *time synchronization*, also achieved in three phases:
  - First, the **MoC$_1$** kernel imposes the *time synchronization constraints* that should be satisfied during the **MoC$_2$** cluster executions (⑦ in Figure 5.4).
  - Second, the **MoC$_1$-MoC$_2$ solver** instances proceed with the execution of the elements contained inside the **MoC$_2$** clusters.
  - Third, when the **MoC$_1$-MoC$_2$** synchronization actions are required, the solver instances return the simulation control to the **MoC$_1$** kernel (⑨ in Figure 5.4), via statements defined in the **MoC$_1$**.

Similarly, interactions between the **DE MoC** and **MoC$_2$** (⑩ in Figure 5.4) could be performed by a **DE-MoC$_2$ solver** (⑫ in Figure 5.4). This means that a synchronization mechanism (⑪ in Figure 5.4) will be defined between the **DE MoC** and **MoC$_2$**.

In SystemC MDVP the implementation of new synchronization mechanisms (⑬ in Figure 5.4), should consider the three phases to be performed between a *master MoC* and a *slave MoC*: first, the *master MoC* will impose, on the *slave MoC*, the *time synchronization constraints* to be satisfied; second, the *slave MoC solver* will execute the simulation; and third, the *slave MoC solver* will interrupt, or send the results to the *master MoC* at the indicated time. This indicates, that the process executing the *slave MoC* will run in the context of the *master MoC* leading to a hierarchization of the MoCs.

The advantage of the synchronization approach is for the system designer, since this approach allows the automatic selection of synchronization mechanisms for the simulation of a model. Although

several synchronization mechanisms are defined in the SystemC MDVP simulator, by means of the available solvers, only the mechanisms best suited to this model will be selected. For each cluster in the model, a pair of *master-slave MoCs* will be detected and used by the simulator to select the solver that will be instantiated on each cluster. This solver will be responsible for the *elaboration*, *simulation* and *synchronization* of the cluster's components.

For the example shown in Figure 5.3, as the **Cluster C1** is handled as a DE module, but contains components (modules and clusters) handled as **MoC$_1$** modules, then, the pair **DE-MoC$_1$** is the *master-slave* pair of MoCs detected for the **Cluster C1**. This pair is used to determine that the **DE-MoC$_1$ solver** will be instantiated on the **Cluster C1**. In consequence, the components of **Cluster C1**, handled by the **MoC$_1$** (*slave MoC*), will be executed following the *time synchronization constraints* imposed by the **DE MoC** (*master MoC*). Similarly, on the **Cluster C2**, it will be instantiated a **MoC$_1$-MoC$_2$ solver**; and on the **Cluster C3**, it will be instantiated a **DE-MoC$_2$ solver**. In SystemC MDVP, as shown in Figure 5.4, the **DE-MoC$_1$ solver** was defined in the **MoC$_1$**, and the **DE-MoC$_2$ solver** and the **MoC$_1$-MoC$_2$ solver** were defined in the **MoC$_2$**.

The detection of MoCs and the instantiation of solvers imply that the hierarchy of clusters, initially detected by the simulator, is transformed in a *hierarchy of solvers*, which will be used for controlling the elaboration and simulation of components in heterogeneous models. For the example shown in Figure 5.3, the hierarchy of solvers constructed is shown in Figure 5.5.

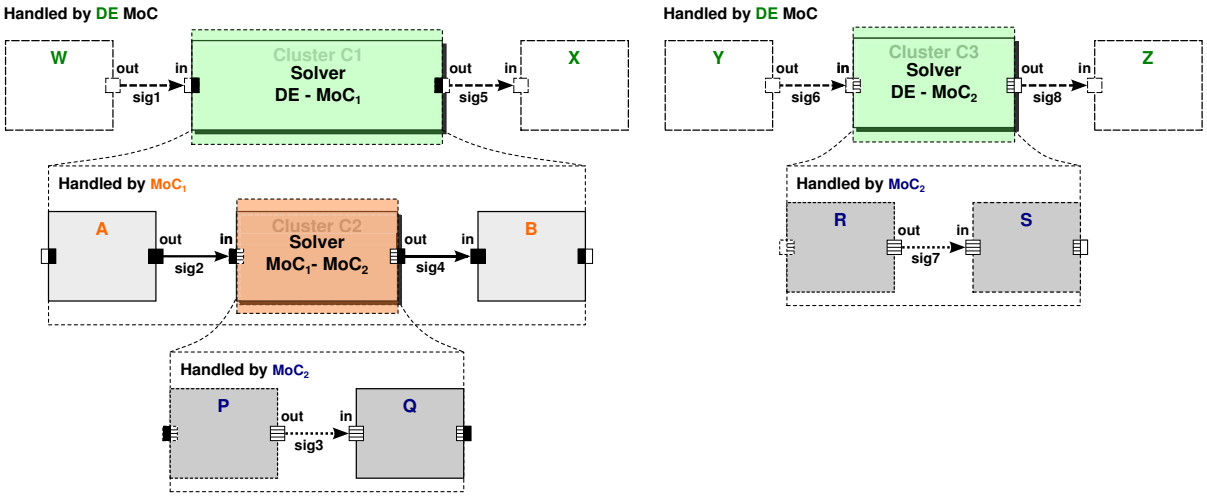

Figure 5.5: Hierarchy of Solvers Constructed from the Hierarchy of Clusters shown in Figure 5.3.

In the hierarchy of solvers, despite that the components of a cluster **C$_i$** are described in different MoCs, they will be handled following a same set of rules called *elaboration and simulation semantics*. These set of rules are defined by the solver instantiated in the cluster **C$_i$**. For the example shown in Figure 5.5, the components of **Cluster C1** (module **A**, module **B** described in the **MoC$_1$**; and **MoC$_1$-MoC$_2$ solver** described in the **MoC$_2$**) will be elaborated and simulated following the rules imposed by the **DE-MoC$_1$ solver**.

In SystemC MDVP, the *elaboration and simulation semantics* will be associated to each particular MoC implemented in the simulator. They will be defined by means of abstract classes called *MoC interfaces*, and implemented by the modules and solvers described in such particular MoCs.

5.4.2. MoC Elaboration and Simulation Semantics

The **elaboration and simulation semantics** associated to a MoC in SystemC MDVP, are abstract methods called by the simulator to perform the *elaboration* and *simulation* phases on a set of modules and solvers instantiated inside a cluster.

To ensure that the SystemC MDVP modules and solvers instantiated in the same hierarchical level are elaborated and simulated under the same rules:

- Each module should implement the *abstract semantics* defined by the MoC in which it is defined.
- Each solver of a MoC_2 should implement the *abstract semantics* defined by the MoC_1 with which their components want to communicate.

For the case of DE modules, as the elaboration and simulation are ensured by the SystemC DE kernel, we define only *DE abstract semantics* for handling the solvers which want to communicate with DE. This corresponds to define a set of abstract methods and encapsulate them in a class called **DE MoC interface**. In the example shown in Figure 5.5, the *DE abstract semantics* correspond to the methods which will trigger the elaboration and simulation phases of the **DE-MoC_1 solver** and the **DE-MoC_2 solver**.

The *DE MoC interface* is defined by means of two abstract methods elaborate() and simulate(). These methods are implemented in different ways according to the semantics of the MoC, which wants to communicate with DE. An example is shown in Figure 5.6.

Assuming that the elaboration and simulation semantics defined by the **MoC_1** correspond to the abstract methods elab_m1() and sim_m1(), and the semantics defined by the **MoC_2** correspond to the abstract methods elab_m2() and sim_m2(), then:

- The elaborate() and simulate() methods implemented in the **DE-MoC_1 solver** call the elab_m1() and sim_m1() methods, respectively, on each one of its components (modules described inside the **MoC_1**, and solvers which want to interacts with the **MoC_1**).
- Similarly, the elaborate() and simulate() methods implemented in the **DE-MoC_2 solver** call the elab_m2() and sim_m2() methods, respectively, on each one of its components (modules described inside the **MoC_2**).
- Using the same approach, the elab_m1() and sim_m1() methods in the **MoC_1-MoC_2 solver**, call the elab_m2() and sim_m2() methods, respectively, on each one of its components (modules described in the **MoC_2**).

In the SystemC MDVP, the implementation of the abstract methods in *solvers* will be performed by the MoC designer when such solvers are created.

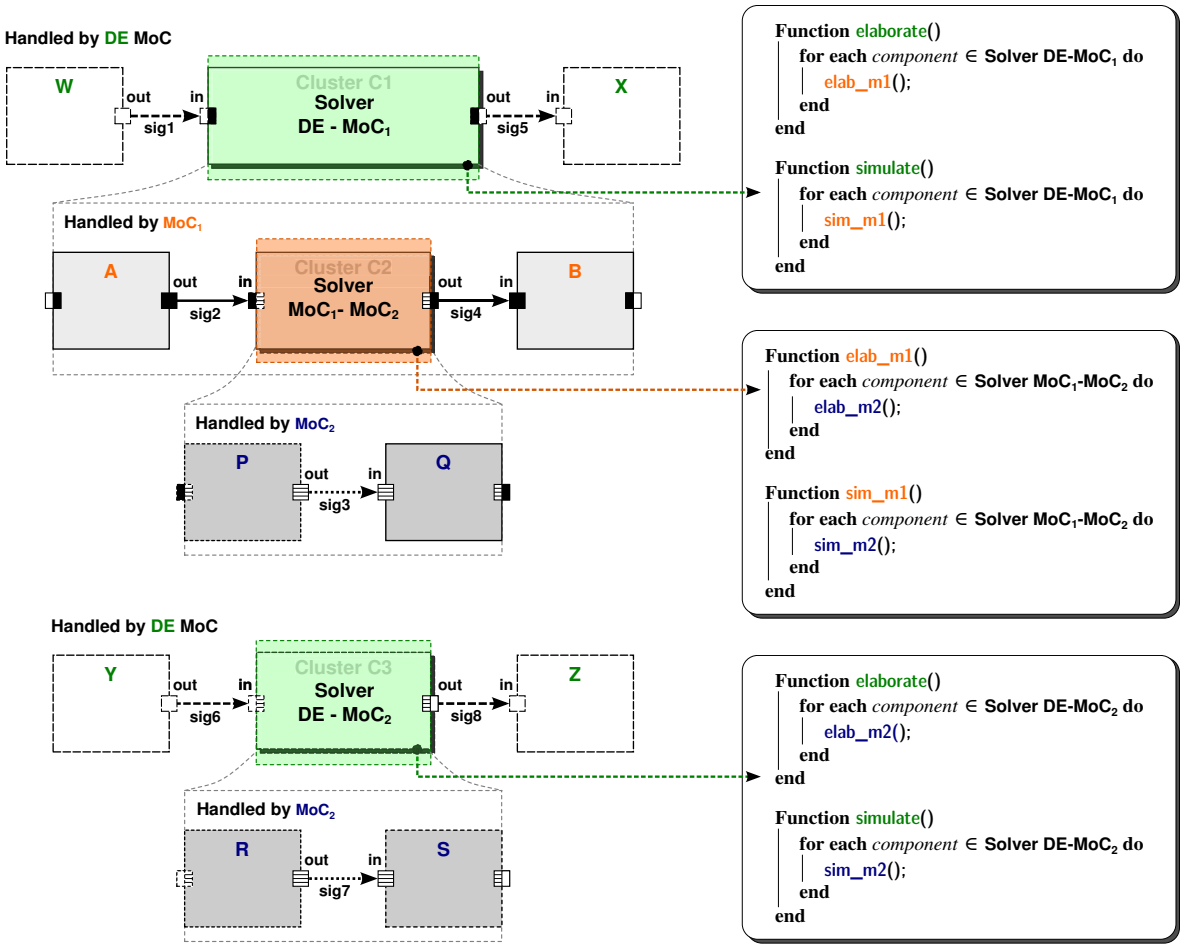

Figure 5.6: Example of the Abstract Elaboration and Simulation Semantics in SystemC MDVP.

In order to detail the steps followed by the simulator, in Section 5.5, we define the generic elaboration and simulation phases of the SystemC MDVP simulator kernel.

5.5. Elaboration and Simulation Phases in SystemC MDVP

When a designer creates a model in SystemC MDVP, and calls the sc_start() method, the model is ready to be analyzed and prepared for simulation.

Because the SystemC MDVP simulation kernel is presented as an extension of the SystemC DE kernel, introduced in Section 2.2.2, the first stage of the model creation is supported by the *SystemC elaboration* phase. This stage is the construction of the module hierarchy, which facilitates the traversal of modules, ports, and channels instantiated in a model.

In SystemC MDVP, based on the semantics defined by the SystemC standard, we extend the *elaboration* and *simulation* phases as shown in Figure 5.7.

On the one hand, during the *SystemC MDVP Elaboration*, we introduce generic methods for performing the identification and creation of clusters; the instantiation of solvers on the created clusters; and the hierarchical elaboration of SystemC MDVP objects (modules, ports and channels) instantiated in a model, regardless of the MoCs to which they belong. These generic methods are executed under the context of the SystemC end_of_elaboration() callback.

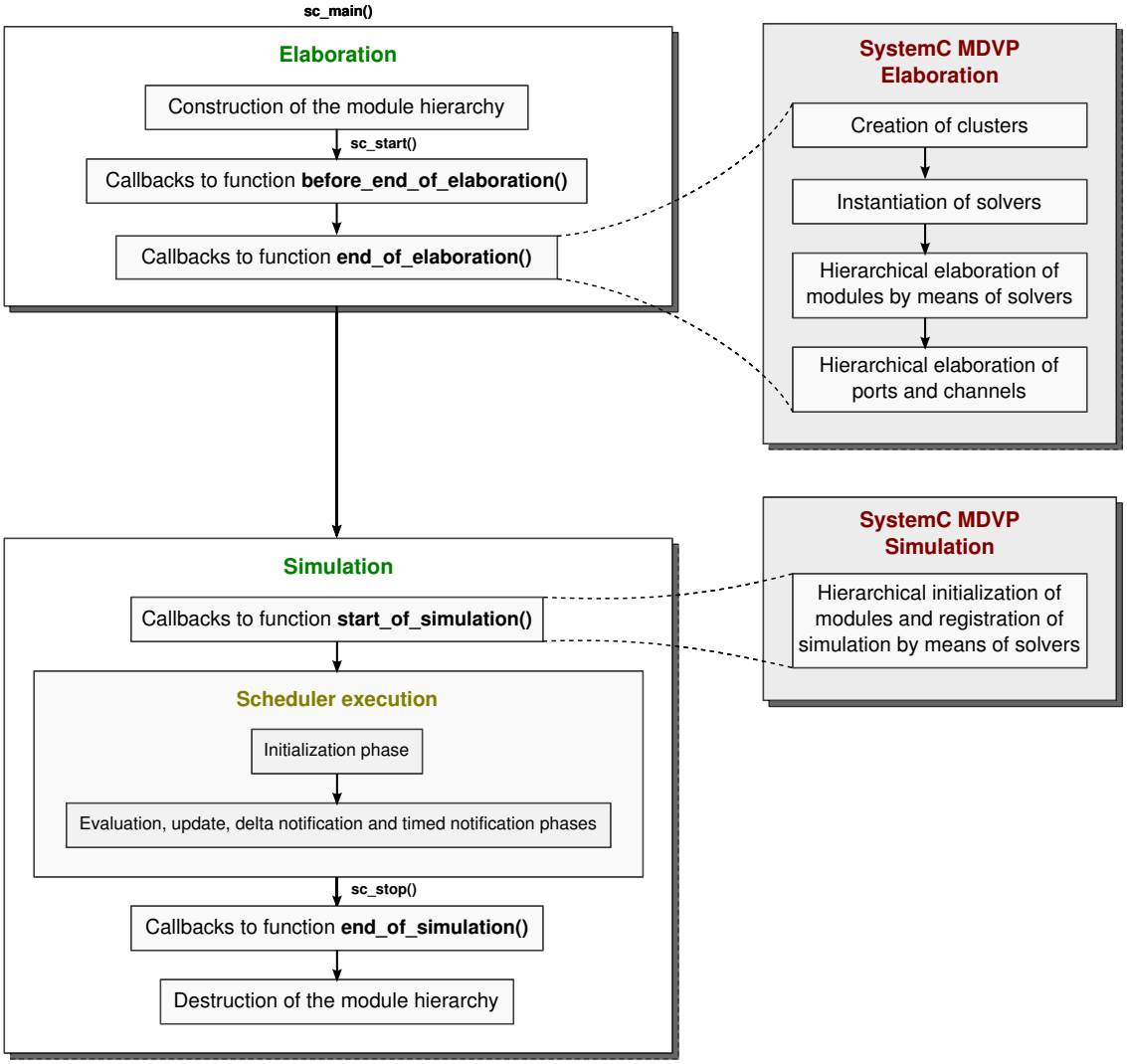

Figure 5.7: SystemC MDVP Elaboration and Simulation Phases.

On the other hand, during the *SystemC MDVP Simulation*, we introduce one generic method for performing the initialization of modules instantiated in a model, and the registration of the clusters' simulation in the DE kernel. This method is executed under the context of the SystemC start_of_simulation() callback.

5.5.1. Elaboration Phase

a. Creation of Clusters

In this stage, a hierarchical view of the model is created by means of the exploration of instantiated modules, ports and channels. During exploration, we identify the different *clusters* of interconnected modules, which belong to a same MoC. These clusters can be considered as homogeneous regions limited by converter ports, which perform the communication between two different MoCs. An example of the cluster identification was shown in Figure 5.2.

Once the *clusters* have been identified, the hierarchical view of the model is constructed by means of a tree data structure. Nodes contained in such tree are objects called **cluster nodes**, which encapsulate

the information associated to each identified *cluster*. This information corresponds to the set of attributes described below:

- **Master MoC**: is the model of computation which imposes the *time synchronization constraints* for the execution of the cluster's components (modules or clusters). It is identified by exploring the converter ports, which limit the cluster. When converter ports are not present in a cluster, the DE MoC is selected by default.
- **MoC**: is the model of computation in which the cluster's components are defined. It is identified by exploring the modules instantiated inside the current cluster.
- **List of modules**: is the structure containing the modules instantiated inside the current cluster.
- **List of cluster nodes**: is the structure containing the clusters identified inside the current cluster.

For the example previously shown in Figure 5.2, the hierarchical view constructed by means of a tree structure, is shown in Figure 5.8, where three *cluster nodes* (**C1**, **C2** and **C3**) are instantiated to encapsulate the information of the model, and one additional *cluster node* (**Master**) is instantiated to encapsulate the clusters to be handled under the *DE semantics* of SystemC.

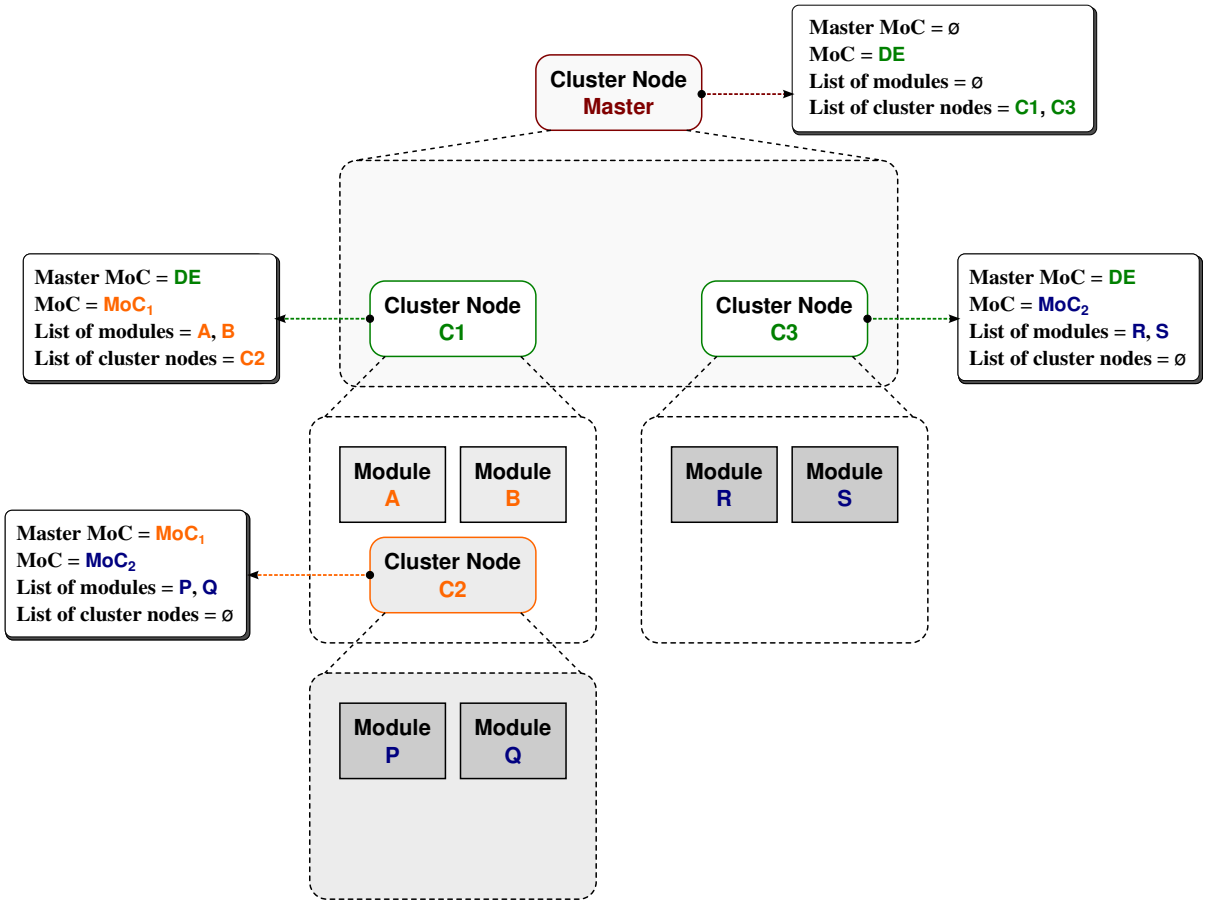

Figure 5.8: Cluster Nodes' Hierarchy of the SystemC MDVP Model shown in Figure 5.2.

The information associated to cluster nodes determines how the components of each cluster will be elaborated and simulated. For the last example, we can deduce that components of **C1** and **C3**, follow the *time synchronization constraints* and implement the elaboration and simulation semantics imposed by the **DE MoC**; and components of **C2**, the semantics imposed by the **MoC_1**.

Once the hierarchical view has been constructed, the pairs of master-slave MoCs which want to interact in the model are also detected. Using these pairs, and the solvers implemented when a MoC is defined, a *dictionary of solvers* is constructed. This dictionary is a structure containing the pair of identified MoCs, and the prototypes of solvers able to handle the interactions between such pair of MoCs. For the example shown in Figure 5.8, the dictionary of solvers corresponds to the structure shown in Table. 5.1. This dictionary will be later used to determine the solver required for the elaboration and simulation of each *cluster node*.

Pair of MoCs < master, slave >	**Solver Prototypes**
< DE, MoC_1 >	DE - MoC_1 solver
< DE, MoC_2 >	DE - MoC_2 solver
< MoC_1, MoC_2 >	MoC_1 - MoC_2 solver

Table 5.1: Dictionary of Solver Prototypes Constructed for the Example shown in Figure 5.8.

The definition and implementation of this stage of *creation of clusters* has been developed in the framework of another thesis work [67], which addresses the compatibility checks of dimensions and units included in a model, its functional verification, and the monitoring and tracing mechanisms that will be also included in the SystemC MDVP simulator prototype.

b. Instantiation of Solvers

In this stage, the solver instances responsible for the elaboration, simulation and synchronization of a model, are created and assigned on the *cluster nodes* previously instantiated.

To this end, the simulator performs a depth-first traversal of the hierarchy of clusters, locates and selects the *clusters nodes* from the bottom to the top of the hierarchy, and executes on each *cluster node* the three steps presented below:

1. **Creating a pair of *master-slave* MoCs**: this pair is created using the attributes associated to the *cluster node*. The master is the MoC imposing the elaboration and simulation semantics (**Master MoC** attribute); and the slave is the MoC in which the cluster's components are defined (**MoC** attribute).

2. **Finding a suitable solver prototype**: the pair of *master-slave* MoCs previously created is found in the dictionary of solvers. Then, the solver prototype associated to this pair of MoCs is selected.

3. **Cloning the solver prototype**: a new solver instance is created by coping the prototype selected from the dictionary of solvers. This new solver instance is assigned on the current *cluster node*.

At the end of this stage, the hierarchical view of the model is converted in a hierarchy of solvers, where each component is a solver instance, with the responsibility of controlling the *elaboration*, *simulation* and *synchronization* of the set of modules and solvers that belong to it. For the example shown in Figure 5.8, the hierarchy of instantiated solvers is shown in Figure 5.9.

c. Hierarchical Elaboration of Modules by means of Solvers

In this stage, each module instantiated by the designer is elaborated. This elaboration is performed on the hierarchy of solvers previously constructed, using the *DE MoC elaboration semantics* (previously defined in Section 5.4.2).

As shown in Figure 5.9, in the first level of the hierarchy, we always have solvers which want to interact with the DE MoC. By definition, these solvers implement the elaborate() method defined in the *DE MoC interface*, which is responsible for the elaboration of the modules and solvers contained in the clusters interacting with DE.

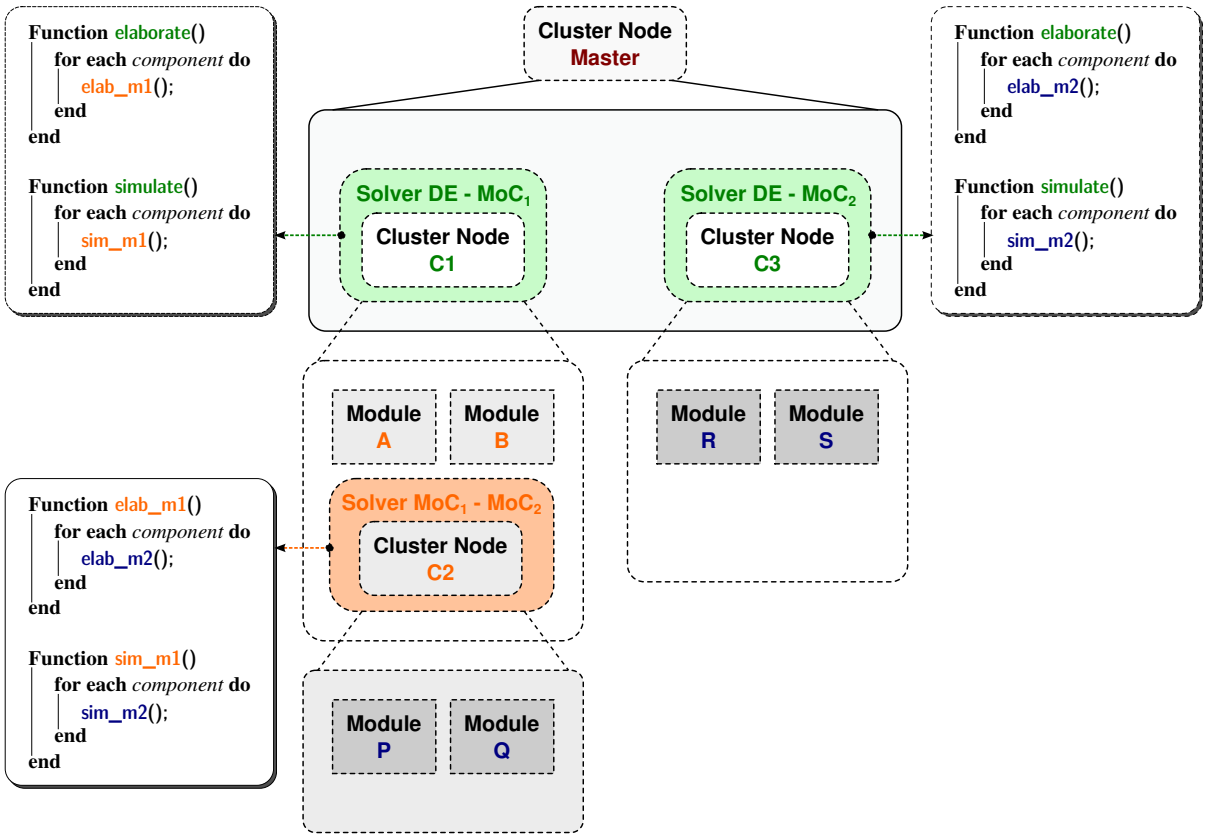

Figure 5.9: Hierarchy of Solvers of the SystemC MDVP Model shown in Figure 5.2.

Therefore, the hierarchical elaboration is defined as a function, which calls the elaborate() methods of the solvers encapsulated in the cluster node **Master**. This results in the call of the elaboration methods defined for each cluster component included in the hierarchy. In the example shown in Figure 5.9:

- When the elaborate() method of the **DE - MoC$_1$ solver** is called, the elaboration method elab_m1() implemented by the module **A**, module **B**, and **MoC$_1$ - MoC$_2$ solver** is automatically performed:
  - In the modules, the elab_m1() method could be defined by the designer.
  - In the solver, such method calls the elaboration method elab_m2() implemented by each one of its components (module **P** and module **Q**).
- Similarly, when the elaborate() method of the **DE - MoC$_2$ solver** is called, the elaboration method elab_m2() implemented by the module **R** and module **S** is automatically performed.

At the end of the stage, thanks to the *MoC elaboration semantics* defined when each MoC is implemented in the simulator, all the modules instantiated by the designer are elaborated. An example of implementation of a MoC and its elaboration semantics is presented in Chapter 6.

d. Hierarchical Elaboration of Ports and Channels

To perform the elaboration of ports and channels, SystemC MDVP imposes the condition that an elaborate() method must be implemented by each port and channel instantiated in a model. The method implementation will be generically specified for each type of port or channel added in the simulator when a MoC is defined.

By default, this method should not be defined by the designer. This means for example, that when a MoC is created, the methods for determining the initial values of ports, or the size of channels, can be encapsulated on elaborate() methods associated to each type of object.

Imposing the previous condition, the hierarchical elaboration of ports and channels is reduced to the stages presented below:

1. Performing a depth-first traversal of the hierarchy of solvers.
2. Locating and selecting the modules or solvers from the bottom to the top of the hierarchy.
3. Accessing to the ports and channels associated to each module or solver.
4. Calling the elaborate() method of each port and channel, which has not been elaborated.

Note: access from a module to a port, and from a port to a channel will be guaranteed by the SystemC MDVP kernel, which takes advantage of the methods provided by SystemC for traversing the hierarchy of modules defined when the SystemC elaboration phase starts.

At the end of the present stage, all ports and channels associated to the modules are elaborated and prepared for the simulation.

5.5.2. Simulation Phase

Following the same approach used during elaboration, the **initialization and registration of the simulation** is performed on the hierarchy of solvers previously constructed, using the *DE MoC simulation semantics* (previously defined in Section 5.4.2).

As previously discussed, in the first level of the hierarchy, we always have solvers which want to interact with the DE MoC. By definition, these solvers implement the simulate() method defined in the *DE MoC interface*, which is responsible of:

- Initializing the modules and solvers contained in the clusters interacting with DE.
- Registering, in the SystemC DE simulation kernel, a simulation thread containing the information required to trigger the simulation of the solvers interacting with DE. This registration creates a *SystemC dynamic process*, by means of the method sc_spawn().

Therefore, the hierarchical initialization and simulation is defined as a function, which calls the simulate() methods of the solvers encapsulated in the cluster node **Master**. This results in the call of the simulation methods defined for each cluster component included in the hierarchy. In the example shown in Figure 5.9:

- When the simulate() method of the **DE - MoC$_1$ solver** is called, the simulation method sim_m1() implemented by the module **A**, module **B**, and **MoC$_1$ - MoC$_2$ solver** is automatically performed:
  - In the modules, the sim_m1() method could be defined by the designer, or by default, implemented by the MoC in which the module is defined.
  - In the solver, such method calls the simulation method sim_m2() implemented by each one of its components (module **P** and module **Q**).
- Similarly, when the simulate() method of the **DE - MoC$_2$ solver** is called, the simulation method elab_m2() implemented by the module **R** and module **S** is automatically performed.

At the end of the stage, thanks to the *MoC simulations semantics* defined when each MoC is implemented in the simulator, all the modules instantiated by the designer are initialized and registered to be simulated. An example of the implementation of a MoC and its simulation semantics is presented in Chapter 6.

5.6. Overview of the SystemC MDVP Kernel Implementation

This section describes how the SystemC MDVP kernel, introduced in previous sections, can be implemented as an extension of the SystemC standard.

5.6.1. Kernel Requirements

The SystemC MDVP kernel needs to fulfill several requirements to successfully ensure the generic elaboration and simulation phases defined in Section 5.5, and allow the addition of MoCs.

- To take advantage of the constructors, hierarchy of modules, and elaboration and simulation callbacks offered by the SystemC kernel, the SystemC MDVP modules, solvers, channels and ports should be implemented as classes directly inherited from the ones implementing the SystemC objects.

- To perform the SystemC MDVP elaboration and simulation phases under generic and recursive methods, modules and solvers should be handled using the same SystemC MDVP object. This object will be called *MoC Interface*.

- To implement traversal hierarchy methods in SystemC MDVP:

  - Each module or solver should offer an access to the ports instantiated inside it.

  - Each channel should offer an access to the ports connected to it.

  - Each port should offer an access to the channel to which it is bound, and to the module which contains it.

- To perform the instantiation of solvers in a model, an abstract method clone() should be implemented by each solver, which is defined when a MoC is added to the SystemC MDVP kernel.

- To ensure the elaboration and simulation of solvers, which want to interact with the DE MoC, an interface to communicate with the DE MoC should be defined. This interface will contain the definition of the abstract methods elaborate() and simulate().

- To ensure the elaboration and simulation of ports and channels, an abstract method elaborate() should be implemented by each specific port or channel, which is defined when a MoC is added to the SystemC MDVP kernel.

- To handle the generic elaboration and simulation phases, regardless of the MoCs included, a class called simulation context should be defined. It will be the bridge between the SystemC DE and the SystemC MDVP simulation kernels.

5.6.2. SystemC MDVP Kernel Classes

Taking into account the requirements defined in the last section, the hierarchy of classes defined for the SystemC MDVP kernel is shown in Figure 5.10. These classes will be the basis for the definition of MoCs in the simulator. More details are presented in Section 5.7.

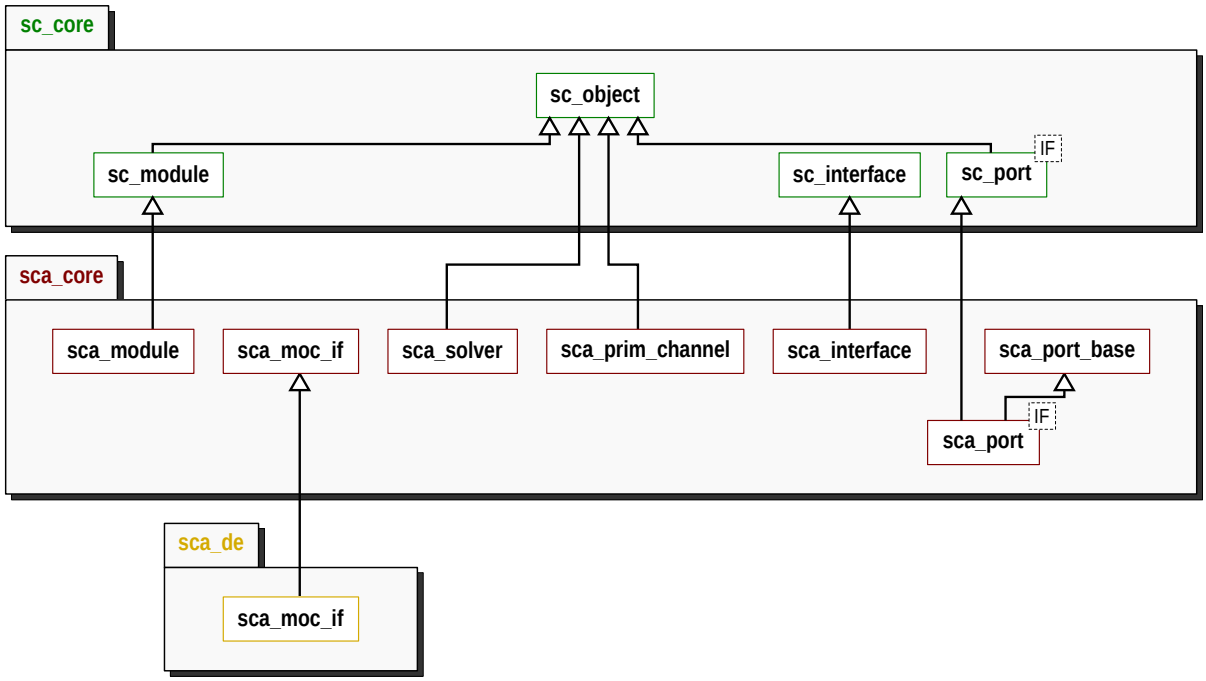

Figure 5.10: Overview of the SystemC MDVP Kernel Classes.

Note: in order to preserve a name compatibility with SystemC-AMS, we use the prefix sca for naming the SystemC MDVP classes.

a. Module, Solver, and MoC Interface Classes (shown in Figure 5.11)

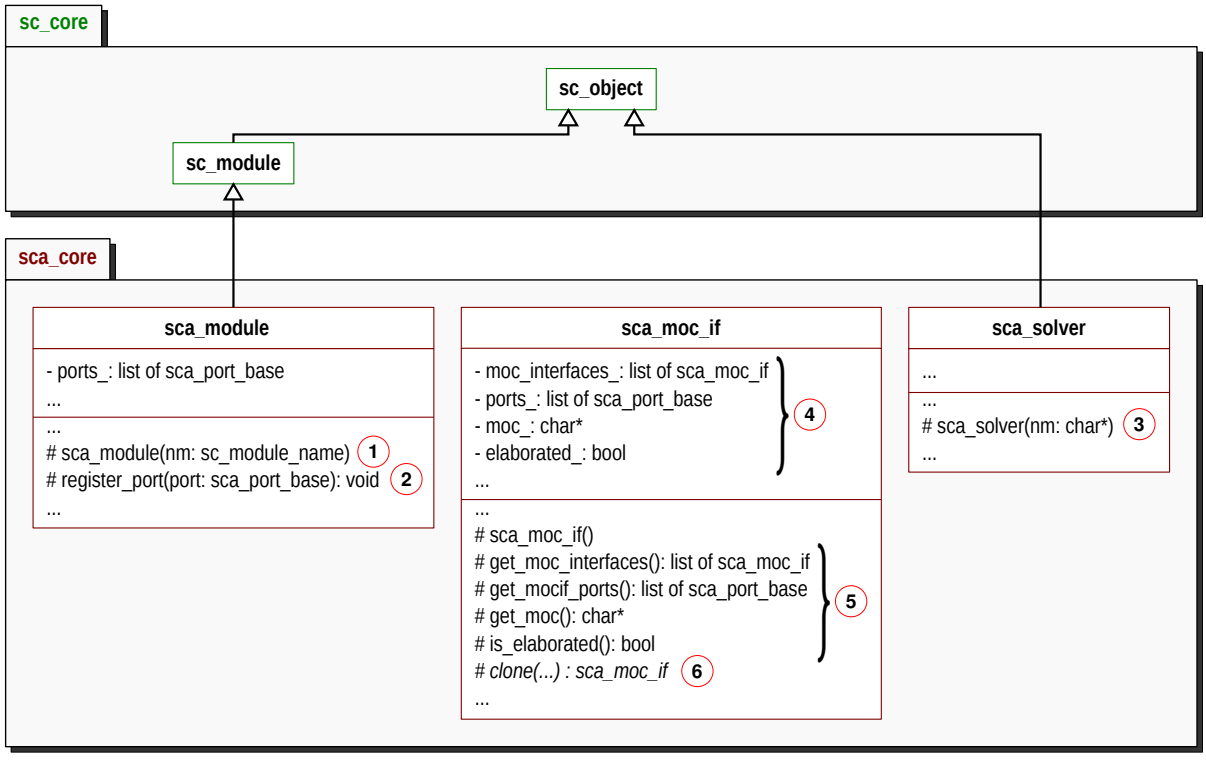

Figure 5.11: Overview of the SystemC MDVP Module, Solver, and MoC Interface Classes.

- The sca_core::sca_module is the base class used for implementing specific modules associated to different MoCs in the simulator. Its constructor (1 in Figure 5.11), in addition of calling the SystemC module constructor, is responsible for registering such module in the SystemC MDVP simulation context. Besides, it offers a method (2 in Figure 5.11) for registering a port within itself. Such registration ensures that a port can be accessed from a module.

- The sca_core::sca_solver is the base class used for implementing specific solvers associated to different MoCs in the simulator. In order to be handled as an element of the hierarchy of objects offered by SystemC, it inherits from the sc_object class. Similarly to the module, thanks to its constructor (3 in Figure 5.11), it is registered in the SystemC MDVP simulation context.

- The sca_core::sca_moc_if is the class created to generically handle the modules and solvers during the SystemC MDVP elaboration and simulation phases. During elaboration, instances of this class will be used to represent the hierarchy of solvers constructed for a model.

  The class attributes (4 in Figure 5.11) indicate that an instance of a sca_core::sca_moc_if, for example the solver instantiated on a cluster node **C1**, can contain:

  - moc_interfaces_: is the list of modules and solvers instantiated inside the current sca_moc_if instance, for example the components of cluster node **C1**.
  - ports_: is the list of ports of the current sca_moc_if instance, for example the converter ports associated to cluster node **C1**.
  - moc_: is the model of computation associated to the current sca_moc_if instance, for example the model of computation associated to cluster node **C1**.
  - elaborated_: is the status of elaboration of the current sca_moc_if instance.

  In addition, several methods (5 in Figure 5.11) are defined to provide the access to the attributes of the current class; and one abstract method called clone() (6 in Figure 5.11) is defined to ensure the stage of *instantiation of solvers*. The last method should be implemented by the MoC designer to return a MoC specific solver instance (see Section 5.5.1.b).

b. Interface and Channel Classes (shown in Figure 5.12)

- The sca_core::sca_interface is the base class used for implementing specific interfaces defined by the different MoCs in the simulator. It is created to generically handle the interfaces included in SystemC MDVP. It inherits the constructor and methods from the SystemC interface class.

- The sca_core::sca_prim_channel is the base class used for implementing the specific channels defined by the different MoCs in the simulator. It defines the common attributes of a channel (1 in Figure 5.12), regardless of the model of computation to which it is associated:

  - ports_: is the list of ports bound to the current channel. Having this list, any channel can access any port connected to it.
  - moc_: is the model of computation associated to a sca_prim_channel instance.
  - elaborated_: is the status of elaboration of a sca_prim_channel instance.

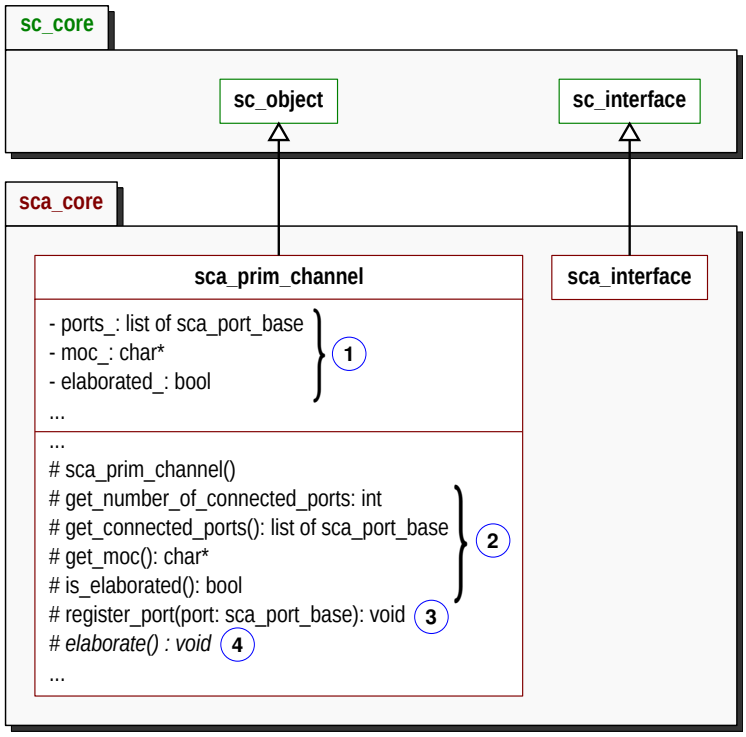

Figure 5.12: Overview of the SystemC MDVP Interface and Channel Classes.

This class in addition of implementing a constructor, which registers each channel in the SystemC MDVP simulation context, includes several methods (2 in Figure 5.12) to provide the access to the attributes of the current class, and one method (3 in Figure 5.12) which performs the registration of a port in a channel.

Moreover, as this class is created to generically handle the channels included in SystemC MDVP, it defines the abstract method elaborate() (4 in Figure 5.12), which should be implemented by the specific channels defined by each MoC in the simulator.

c. Port Classes (shown in Figure 5.13)

- The sca_core::sca_port_base is the base class created to generically handle the ports in SystemC MDVP, regardless of its type, implemented interface, or MoC in which they are defined. It defines the common attributes required to identify a port during elaboration or simulation. These attributes (1 in Figure 5.13) are presented below:
  - connected_ports_: is the list of ports connected to the current port. They are stored to ease the traversal of the hierarchy.
  - moc_: is the model of computation associated to a sca_port_base instance.
  - conversion_moc_: in the case of converter ports, it is the model of computation to which a sca_port_base instance wants to communicate.
  - input_, output_ and converter_: are the attributes indicating the type of a sca_port_base instance. This attributes should be initialized when a port is constructed in a specific MoC.
  - elaborated_: is the status of elaboration of a sca_port_base instance.

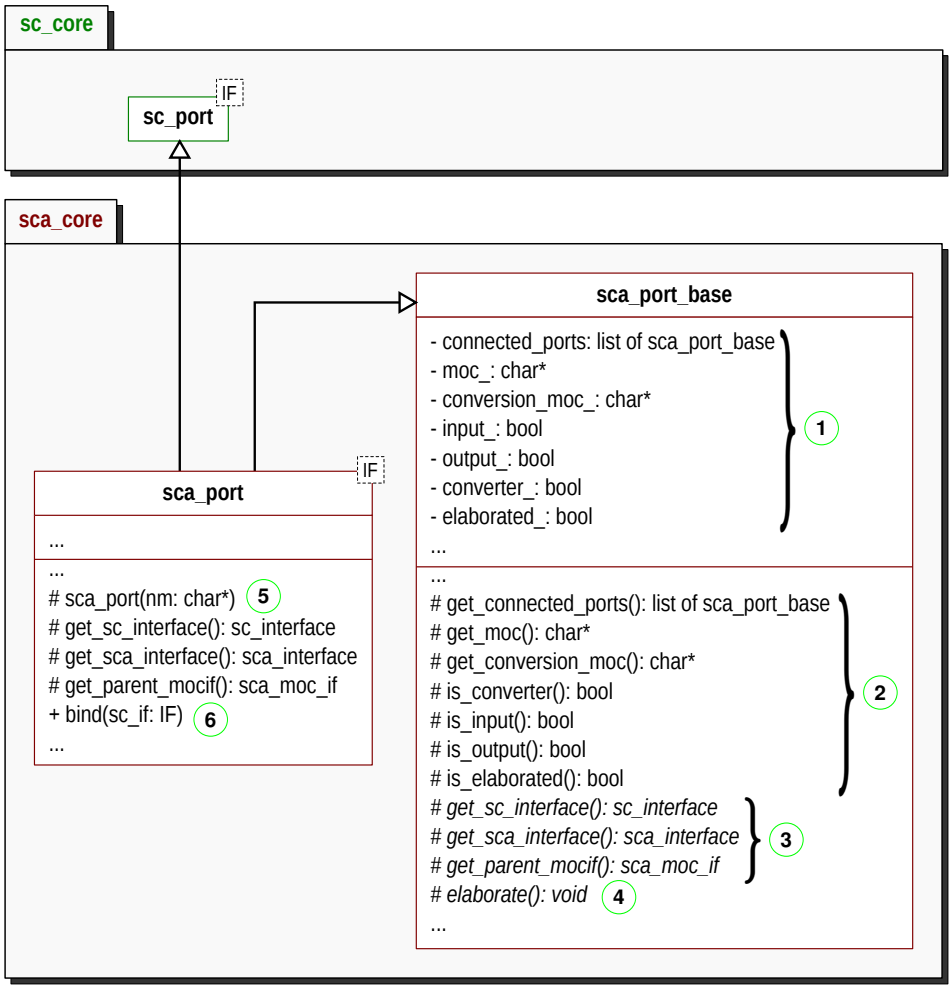

Figure 5.13: Overview of the SystemC MDVP Port Classes.

Similarly to channels, a set of methods (2 in Figure 5.13) are defined to provide the access to the port attributes. Other set of abstract methods (3 in Figure 5.13) are defined to guarantee the access from a port to a channel, or from a port to the MoC interface (module or solver) which contains it.

Moreover, as this class is created to generically handle the ports included in SystemC MDVP, it defines the abstract method elaborate() (4 in Figure 5.13), which should be implemented by the specific ports defined by each MoC in the simulator.

- The sca_core::sca_port<IF> is the base class defined for implementing specific ports in the simulator. It inherits the methods defined by the sc_port class, and the attributes and methods defined by the sca_port_base class. It also implements the abstract methods defined in the sca_port_base class.

  The constructor of this class (5 in Figure 5.13) is responsible for registering the port instance in the module which contains it. This is possible because SystemC provides a method to get the parent object of a port (the module which contains it), and our class sca_module provides a method register_port() which can be used to this end.

  Moreover, this class overloads the bind() method (6 in Figure 5.13), for registering the port instance, in the channel to which it will be bound.

d. DE MoC Interface Class

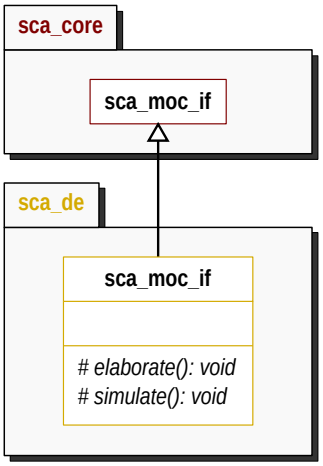

Figure 5.14: Overview of the SystemC MDVP DE MoC Interface Class.

The sca_de::sca_moc_if is the class which defines the interface for communicating with the DE MoC. This interface, as defined in Section 5.4.2, includes the abstract methods elaborate() and simulate(), called by the simulator to perform the elaboration and simulation phases of the solvers which want to interact with the DE MoC.

5.6.3. SystemC MDVP Kernel Implementation Details

In this section we introduce the hierarchy of classes used by the simulator to perform the elaboration and simulation phases defined in Section 5.5. This hierarchy is shown in Figure 5.15.

- The sca_core::detail::sca_simcontext is the class which controls the call to the elaboration and simulation phases in SystemC MDVP. As only one object of this class will be instantiated per simulation, it is implemented using a *singleton* creational design pattern [68].

  Via the implementation of the end_of_elaboration callback (1 in Figure 5.15), this class performs the elaboration phase defined in Section 5.5.1. In the class, specific methods (3 in Figure 5.15) are defined for each one of the stages accomplished during elaboration.

  Via the implementation of the start_of_simulation callback (2 in Figure 5.15), this class performs the simulation phase defined in Section 5.5.2. As for the elaboration, a specific method (4 in Figure 5.15) is defined for the stage accomplished during simulation.

- The sca_core::detail::sca_cluster_node is the class used for defining the cluster nodes that should be encapsulated in a hierarchy of clusters, during the SystemC MDVP elaboration phase. A cluster node, in addition to have the four attributes introduced in Section 5.5.1.a: master_moc_, moc_, moc_ifs_ (list of modules) and nodes_ (list of cluster nodes); it has an attribute moc_interface_, to store the solver instance, which is responsible for the synchronization between the pair of MoCs associated to the current cluster. The access to the cluster nodes attributes (5 in Figure 5.15), is guaranteed by means of the set of methods also defined in this class (6 in Figure 5.15).

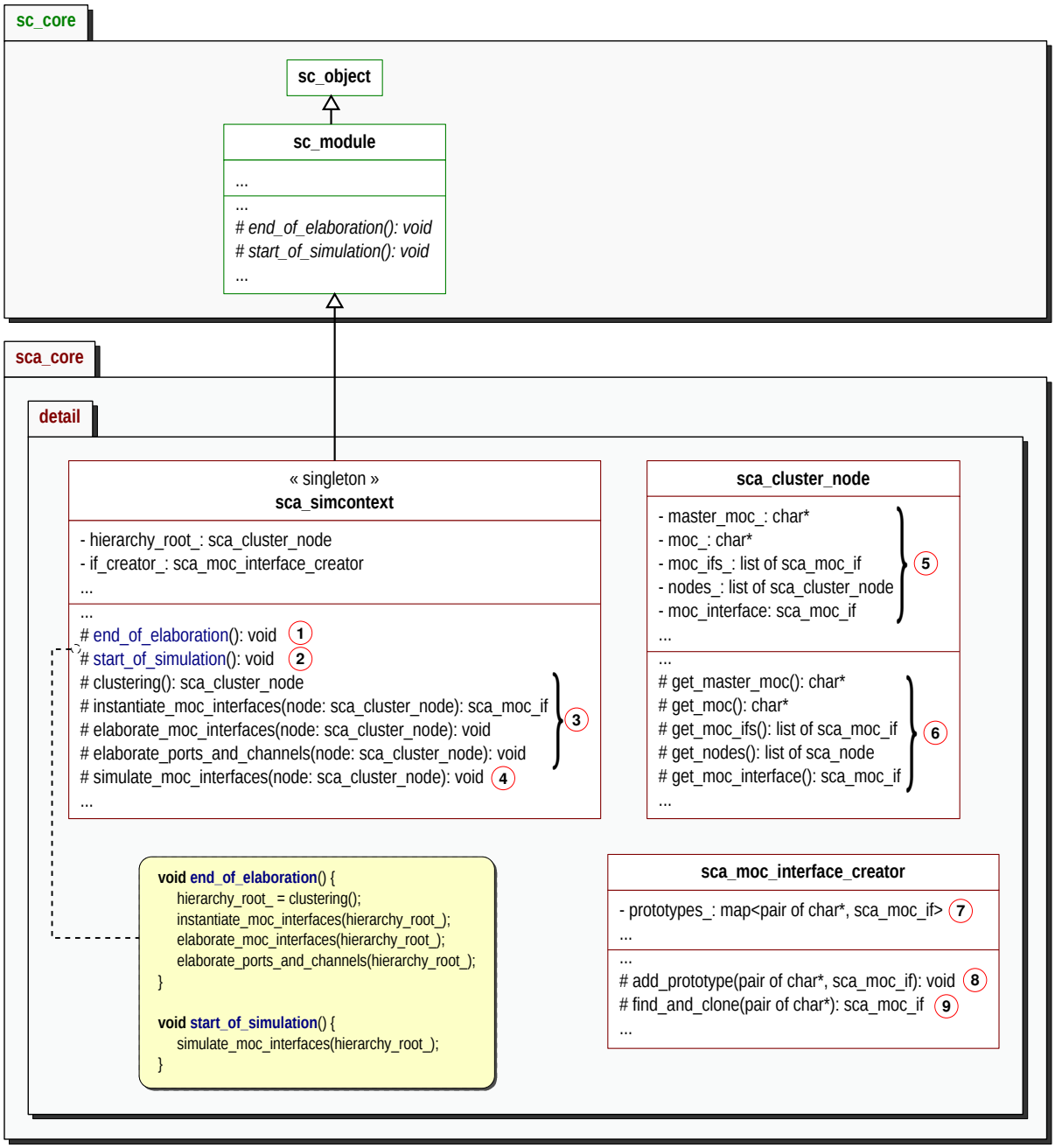

Figure 5.15: Overview of the SystemC MDVP Implementation Details.

- The sca_core::detail::sca_moc_interface_creator is the class created to handle the instantiation of solvers in a model. It can be considered as the factory of solver prototypes, which is implemented following a *prototype* creational design pattern [68]. It contains:
  - The dictionary of solver prototypes that can be instantiated in a model (7 in Figure 5.15).
  - The method add_prototype() (8 in Figure 5.15), which allows the simulator to add new solver prototypes in the dictionary.
  - The method find_and_clone() (9 in Figure 5.15), which receives a pair of MoCs, searches this pair in the dictionary of solver, locates the solver associated to the pair of MoCs, and calls the clone() method implemented by the located solver.

5.6.4. SystemC and SystemC MDVP Interconnection

As introduced in Section 2.2.2, four callbacks (see Figure 2.4) are automatically executed during the elaboration and simulation phases of SystemC for allowing the applications to perform further elaboration and simulations actions. These callbacks are abstract methods, which can be overloaded by SystemC objects or object derived from them.

As the sca_simcontext class (SystemC MDVP simulation context) inherits from the sc_core::sc_module class, the interconnection between the SystemC and SystemC MDVP kernels, and the automatic execution of the elaboration and simulation phases in the simulator, can be easily performed:

- SystemC MDVP elaboration is encapsulated in the end_of_elaboration() callback.
- SystemC MDVP simulation is encapsulated in the start_of_simulation() callback.

5.7. Methodology to Add Models of Computation in SystemC MDVP

In order to add a MoC in SystemC MDVP, a set of classes should be implemented to allow the modeling under a specific time abstraction, and particular computation, communication, synchronization, elaboration and simulation semantics, as previously introduced in Section 5.2. To this end, the requirements presented below should be considered.

- **Define the MoC Interface**: it is the specification of the set of abstract methods allowing the elaboration and simulation of:
  - Modules described in the MoC being defined.
  - Solvers created to communicate with the MoC being defined.
- **Provide the designer with specific MoC components**: it is the specification of modules, ports and channels classes, which inherit from the SystemC MDVP kernel classes previously presented in Figure 5.10.
- **Locate the MoC inside the SystemC MDVP architectural model**: as described in Section 5.4.1, a MoC can be implemented in a particular hierarchical level, according to the desired interactions. When a MoC is located under one of the existing MoCs, it should define:
  - *Conversion ports* to handle the *data synchronization* between the MoC being defined, and the one over it.
  - A *solver* able to handle the elaboration and simulation of the MoC components, and the *time synchronization* between the MoC being defined, and the one over it.

In this section, by means of generic examples, we introduce a methodology for adding MoCs in SystemC MDVP. We show how a MoC can be added at different levels in the hierarchy of classes, and how it can be defined to directly communicate with the DE MoC, or with any other MoC. We have simplified the task of adding MoCs by means of four phases: *addition of MoC's modules*, *addition of MoC's channels*, *addition of MoC's ports*, and *addition of MoC's solvers*.

5.7.1. Addition of MoC's Modules

In order to implement the objects responsible for encapsulating the behaviors associated to a particular MoC_1, two classes should be defined as shown in Figure 5.16.

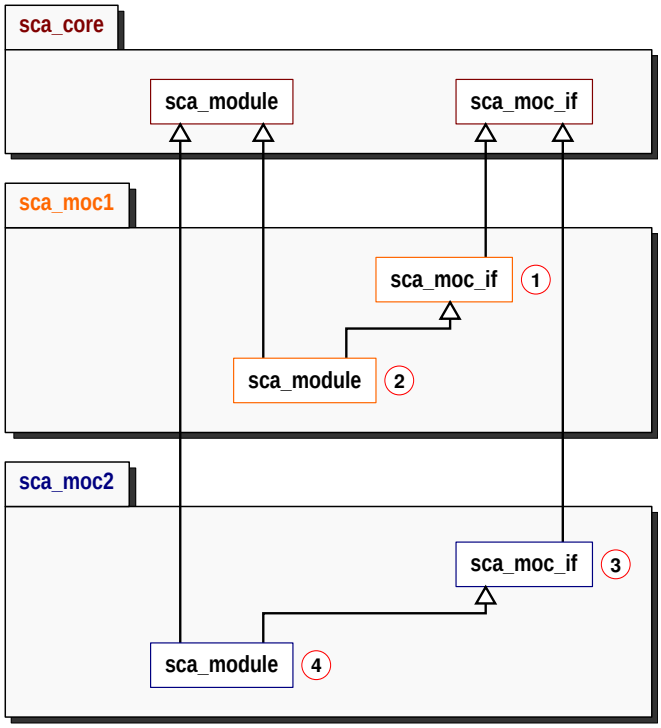

Figure 5.16: Overview of the Addition of MoC's Modules in SystemC MDVP.

- The sca_moc1::sca_moc_if class (1 in Figure 5.16), which is created to generically handle the components of MoC_1 clusters (*modules* belonging to MoC_1, and *solvers* which want to interact with MoC_1). This class should inherit the attributes and methods defined by the sca_core::sca_moc_if class; and define the *MoC interface* of MoC_1, this means, define the abstract methods to be called during the elaboration and simulation of MoC_1 clusters.

  In addition, this class should define the attributes of the MoC_1 modules, and it should define and implement the methods to be generically called in components belonging to MoC_1 clusters. By default, the constructor of this class should initialize the moc_ and elaborated_ attributes inherited from the sca_core::sca_moc_if class.

- The sca_moc1::sca_module class (2 in Figure 5.16), which is provided to the designer to be inherited or instantiated in a model. In the case where this is an abstract class, the designer can inherit from it to represent a block with a particular behavior. Otherwise, the designer can directly instantiate it, because it represents a predefined block (primitive). In this case, all the abstract methods defined in the sca_moc1::sca_moc_if class are implemented. This class should inherit the attributes and methods defined by the sca_core::sca_module class, and the sca_moc1::sca_moc_if class.

Regardless of the location of a MoC in the *SystemC MDVP architectural model*, MoC modules should be always implemented following the description previously presented. For example, the implementation of modules belonging to a MoC_2 is summarized by the definition of:

- The sca_moc2::sca_moc_if class (③ in Figure 5.16), which inherits the attributes and methods from the sca_core::sca_moc_if class, defines the *MoC interface* of MoC_2, defines the attributes of MoC_2 modules, and defines and implements the methods to be generically called on components belonging to MoC_2 clusters.

- The sca_moc2::sca_module class (④ in Figure 5.16), which is provided to the designer to be inherited or instantiated in a model; and inherits the attributes and methods defined by the sca_core::sca_module and the sca_moc2::sca_moc_if classes.

5.7.2. Addition of MoC's Channels

In order to implement predefined channels associated to a particular MoC_1, several classes should be defined as shown in Figure 5.17.

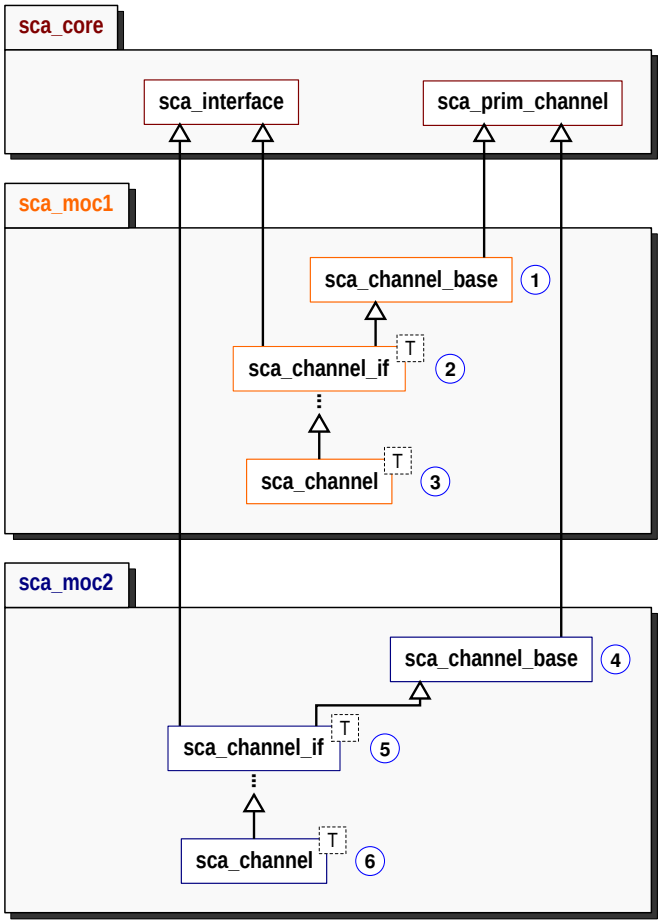

Figure 5.17: Overview of the Addition of MoC's Channels in SystemC MDVP.

- The sca_moc1::sca_channel_base class (① in Figure 5.17), which is created to generically handle the channels defined in MoC_1. This class should inherit the attributes and methods defined by the sca_core::sca_prim_channel class; initialize the inherited moc_ and elaborated_ attributes; and implement the inherited elaborate() abstract method, which will be called during the stage of *elaboration of ports and channels* introduced in Section 5.5.1.d.

- The sca_moc1::sca_channel_if<T> class (2 in Figure 5.17), which defines the interface to be implemented by a specific port. It implements the data structure containing the information of type T transmitted between modules, and defines the methods to be called on such data structure (e.g. read() and write() methods). This class can be decomposed in several classes when the MoC architect wants to provide different interfaces (e.g. input and output interfaces) for the implemented ports. It should inherit from the sca_core::sca_interface and the sca_moc1::sca_channel_base classes.

- The sca_moc1::sca_channel<T> class (3 in Figure 5.17), which is provided to the designer to be instantiated in a model. This class should inherit the attributes and methods defined by the sca_core::sca_channel_if<T> class.

Regardless of the location of a MoC in the *SystemC MDVP architectural model*, channels belonging to each MoC should be always implemented following the description previously presented. For example, the implementation of a channel belonging to a MoC_2 is summarized by the definition of the classes: sca_moc2::sca_channel_base (4 in Figure 5.17), sca_moc2::sca_channel_if<T> (5 in Figure 5.17), and sca_moc2::sca_channel<T> (6 in Figure 5.17).

These classes follow the same description and the same inheritance rules presented for the implementation of the MoC_1 channels.

5.7.3. Addition of MoC's Ports

In order to implement predefined ports associated to a particular MoC_1, several classes should be defined as shown in Figure 5.18.

- The sca_moc1::sca_port_base class (1 in Figure 5.18), which is created to generically handle the ports defined in the MoC_1. This class should inherit the attributes and methods defined by the sca_core::sca_port_base class, initialize the inherited moc_ and elaborated_ attributes; and implement the inherited elaborate() abstract method, which will be called during the stage of *elaboration of ports and channels* introduced in Section 5.5.1.d. In addition, this class defines and initializes the attributes of the MoC_1 ports, and implements the methods of the MoC_1 ports, which do not depend on the port type.

- The sca_moc1::sca_port<IF,T> class (2 in Figure 5.18), which implements functions responsible for calling the methods defined by the interfaces (e.g. read() and write()). This class should inherit the attributes and functions defined in the sca_core::sca_port<IF> and sca_moc1::sca_port_base classes.

- The sca_moc1::sca_in<T> and sca_moc1::sca_out<T> classes (3 in Figure 5.18), which represent the MoC_1 classical ports provided to the designer to be instantiated in a module described in a MoC_1. Each one of these classes implements a particular interface (sca_moc1::sca_moc_if<T>) and provides to the designer the methods to access the information contained in the channel associated to each port. These classes should inherit from the sca_moc1::sca_port<IF,T>, implement the interface IF desired, and initialize the inherited input_, output_, and converter_ port attributes.

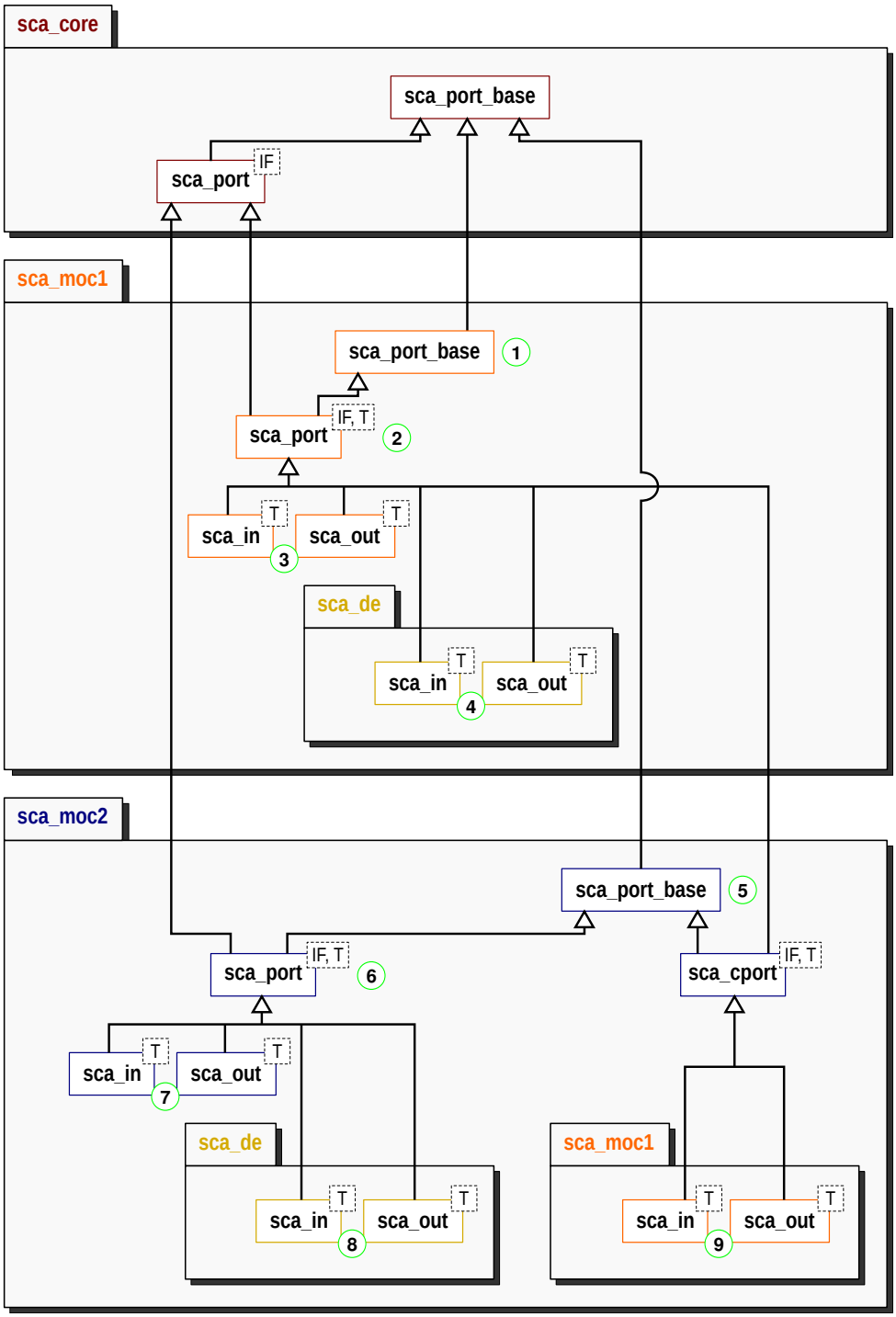

Figure 5.18: Overview of the Addition of MoC's Ports in SystemC MDVP.

- The sca_moc1::sca_de::sca_in<T> and sca_moc1::sca_de::sca_out<T> classes (4 in Figure 5.18), which represent the MoC_1 – DE converter ports provided to the designer, to be instantiated in modules defined in the MoC_1. Each one of these classes implements a particular DE interface (eg. sc_signal_in_if<T> or sc_signal_inout_if<T>), provides to the designer the methods required to access the information there contained, and implements functions for calling the methods defined by the DE interfaces (to have the access to the DE channels). In these classes, the *data synchronization* between the MoC_1 and the DE MoC should be ensured.

If necessary, these ports can overload the elaborate() function for implementing elaboration semantics particular to the converter ports which want to communicate with DE. In addition these classes should initialize the inherited attributes converter_ and conversion_moc_.

The description previously presented should be respected for adding the ports of a MoC, which wants to interact with the DE MoC. For example, the implementation of the ports belonging to a MoC_2, located under the DE MoC in the *SystemC MDVP architectural model*, is summarized in the definition of the classes: sca_moc2::sca_port_base (5 in Figure 5.18), sca_moc2::sca_port<IF,T> (6 in Figure 5.18), sca_moc2::sca_in<T> and sca_moc2::sca_out<T> (7 in Figure 5.18), and sca_moc2::sca_de::sca_in<T> and sca_moc2::sca_de::sca_out<T> (8 in Figure 5.18). These classes follow the same description and the same inheritance rules presented for the implementation of the ports in the MoC_1 (located under the DE MoC).

Another possibility provided by our approach, is the addition of converter ports which want to interact with any other MoC already defined in the simulator. For example, if MoC_2 wants to interact with MoC_1, specific converter ports are required. In this case, as shown in Figure 5.18, the $MoC_2 - MoC_1$ converter ports (9 in Figure 5.18) should be defined as classes inherited from the sca_moc1::sca_port<IF,T> and sca_moc2::sca_port_base. In this way, such classes will have the access to the functions, which handle the methods implemented by the MoC_1 channels connected to the $MoC_2 - MoC_1$ converter ports, and the access to the attributes and particular functions defined for the MoC_2 ports.

5.7.4. Addition of MoC's Solvers

The addition of a solver in the simulator depends on the pair of *master-slave* MoCs, which such solver expects to handle. The *master* MoC is the model of computation which imposes the time synchronization constraints, and the *slave* MoC is the model of computation in which the solver is implemented. Two conditions have to be fulfilled for such implementation:

- The solver inherits from the base class sca_core::sca_solver.
- The solver implements the abstract methods defined in the master's *MoC interface*. These are the methods called to perform the elaboration and simulation of components defined in the slave MoC.

These conditions are illustrated by means of Figure 5.19, where:

- The sca_moc1::detail::sca_de_solver class (1 in Figure 5.19), handles the interactions between the MoC_1 and the DE MoC. It inherits the methods from the sca_core::sca_solver class, and implements the abstract methods (elaborate() and simulate()) defined in the sca_de::sca_moc_if class.
- The sca_moc2::detail::sca_de_solver class (2 in Figure 5.19), handles the interactions between the MoC_2 and the DE MoC. It inherits the methods from the sca_core::sca_solver class, and implements the abstract methods (elaborate() and simulate()) defined in the sca_de::sca_moc_if class.
- The sca_moc2::detail::sca_moc1_solver class (3 in Figure 5.19), handles the interactions between the MoC_2 and the MoC_1. It inherits the methods from the sca_core::sca_solver class, and implements the abstract methods defined in the sca_moc1::sca_moc_if class.

Note: Implementation of solvers is performed following the elaboration and simulation semantics defined in Section 5.4.2.

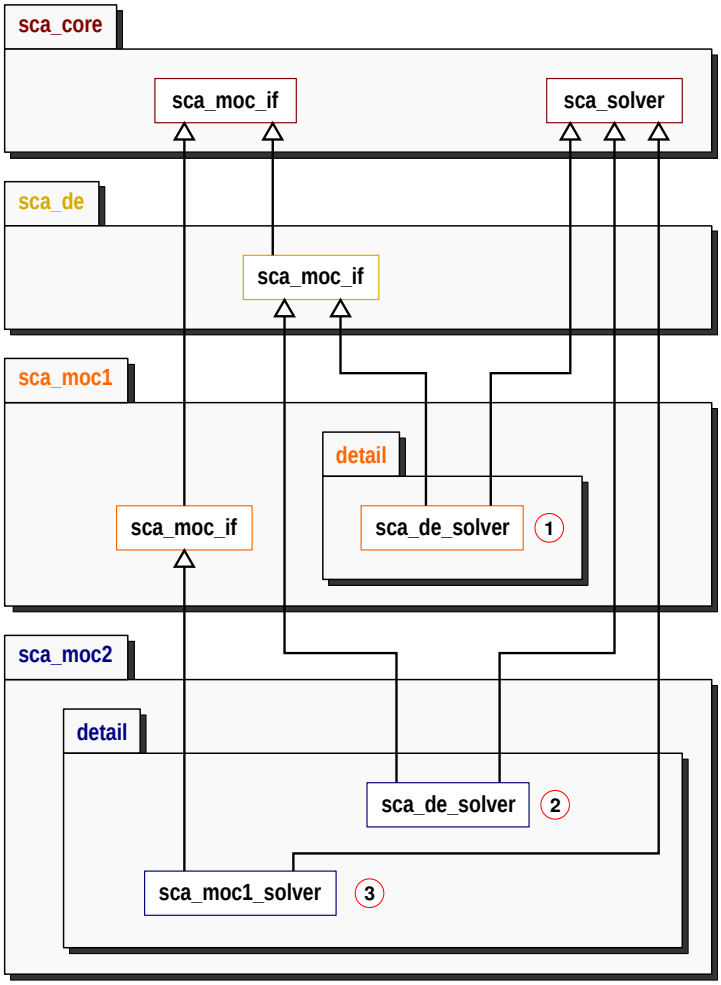

Figure 5.19: Overview of the Addition of MoC's Solvers in SystemC MDVP.

When modules, channels, ports and solvers have been defined, the model of computation is ready to be used by the designer.

5.8. Conclusion and Outlook

After introducing in this chapter the approach followed to describe models in the SystemC MDVP simulator prototype, we presented the synchronization principles, which ensure the *hierarchical modeling* and the implementation of the simulator's *heterogeneity at the kernel-level*. Heterogeneity is implemented by means of well-separated, and hierarchically organized models of computation.

Models of computation provide the set of modules, channels and ports, which can be instantiated and interconnected by the designer in order to describe a model. The simulator identifies the model's clusters, determines the *master-slave relation* associated to each cluster, and automatically selects the solver to be instantiated on each identified cluster. This means that, in the simulator, the designer is neither responsible for implementing nor instantiating the elements, which handle the elaboration, simulation and synchronization of the model's components.

Additionally, we defined generic elaboration and simulation phases for the simulator, which are automatically executed under the control of the SystemC DE kernel. These phases perform the elaboration and simulation methods selected for each model's cluster, and ensure that *the addition of a new model of computation does not modify the SystemC MDVP simulator kernel.*

We introduced how the SystemC MDVP simulator kernel is implemented as an extension of SystemC. We described the base classes for handling modules, channels, ports and solvers during the elaboration and simulation phases. *The implementation of these classes does not modify the DE kernel.*

Finally, we introduced a methodology to add models of computation in the SystemC MDVP simulator prototype. In this methodology, the MoC designer should define the abstract methods which allow the MoC elaboration and simulation; should specify the MoC components to be instantiated by the system designer; should select the master MoC with which the MoC being defined wants to interact; and should implement the converter ports and solver responsible for handling the *data synchronization* and *time synchronization* between the MoC being defined and the master MoC previously selected.

In order to validate the methodology introduced for adding a MoC in SystemC MDVP, in Chapter 6, we present a simplified version of the TDF MoC described in the SystemC AMS standard [13], which directly interacts with the DE MoC. We detail how the TDF elaboration and simulation phases are defined, and how the solver, performing the DE-TDF interactions, is implemented.

CHAPTER

6 Timed Data Flow (TDF) Model of Computation (MoC) in SystemC MDVP

Contents

6.1. Introduction

In this chapter, we define and implement a Timed Data Flow (TDF) Model of Computation (MoC) in the SystemC Multi-Disciplinary Virtual Prototyping (MDVP) simulator prototype. This MoC respects the Discrete Time (DT) semantics, and the computation and communication rules introduced by the SystemC AMS standard [13]. Besides, it includes the synchronization method proposed in Chapter 4, for ensuring the interactions between the Discrete Event (DE) and DT domains.

In Section 6.2, we introduce the requirements to be considered for adding the TDF MoC in SystemC MDVP: the definition of the TDF MoC interface; the specification of the TDF MoC components available for the designer (modules, ports and channels); and the selection of the hierarchical level, where the TDF MoC is located inside the SystemC MDVP architectural model.

In Section 6.3, we define the TDF elaboration and simulation phases, which are automatically called by the SystemC MDVP simulator kernel. We describe how the TDF attributes are assigned in TDF modules and ports, how the TDF clusters are analyzed and initialized, and how the TDF cluster execution is registered in the SystemC DE simulation kernel.

In Section 6.4, we present an overview about the implementation of the TDF MoC. We describe the classes created to represent the TDF simulation objects, and the methods allowing the elaboration and simulation phases in the TDF MoC.

In Section 6.5, by means of an illustrative example, we show the advantages offered by the TDF MoC included in SystemC MDVP.

Finally, in Section 6.6, we conclude this chapter discussing the TDF MoC implementation.

6.2. Requirements for the TDF MoC Implementation

6.2.1. Definition of the TDF MoC Interface

The first requirement to be considered for implementing a MoC in SystemC MDVP is the *definition of the MoC interface*. As described in Chapter 5, the MoC interface is the set of abstract methods allowing the elaboration and simulation of modules described in the MoC being defined, and solvers created to communicate with the MoC being defined.

According to the SystemC AMS standard, as introduced in Section 2.3.2, to describe a TDF module, three functions can be implemented by the designer: set_attributes(), used for fixing the TDF module and port attributes during elaboration; initialize(), used for fixing initial sample values in TDF ports during simulation; and processing(), used for implementing the function, which describes the behavior of the module. In SystemC MDVP, as the solvers which want to interact with the TDF MoC will be handled as TDF modules, the set_attributes(), initialize(), and processing() functions should be implemented by the MoC designer. Therefore, these three functions will be included in the *TDF MoC Interface*.

In addition, we should include in the *TDF MoC interface* the methods required to generically prepare the execution of modules and solvers included in TDF clusters (e.g., methods for handling and verifying the time step relations between modules and ports). These required methods will be identified in Section 6.3.

6.2.2. Specification of the TDF MoC Components

The second requirement to be considered for implementing a MoC in SystemC MDVP is the *specification of the TDF MoC components* offered to the designer for modeling applications in the DT domain. These components are the TDF modules, TDF channels and classical TDF ports, which provide a set of member functions matching the semantics defined by the SystemC AMS standard [13].

a. TDF Modules

The specification of a TDF module in SystemC MDVP is the responsibility of the designer. A base class sca_tdf::sca_module should provide the abstract method processing(), by means of which the module behavior will be implemented; and provide the access to the methods, which set and get the module attribute *time step* (*Tm*). Implementation details of the class sca_tdf::sca_module are presented in Section 6.4.1.

b. TDF Channels

The specification of a TDF channel in SystemC MDVP is not the responsibility of the designer. In the TDF MoC, one predefined channel should be available, by means of the class sca_tdf::sca_signal, to be directly instantiated in TDF models. Implementation details of this class are presented in Section 6.4.2.

TDF channels may be connected to one or more TDF ports, as long as the conditions presented below are satisfied:

- The TDF channel is connected at least from a TDF output port to a TDF input port.
- The TDF channel is connected from one, an only one, TDF output port.
- The TDF channel is connected to one or several TDF input ports.

This means, that only one TDF output port can write information inside the predefined channel, but several TDF input ports can read information from the predefined channel at the same time.

The predefined channel should be specified as an abstract data type, which contains circular data buffers used to temporarily store the information transmitted through the channel. An example is shown in Figure 6.1.

- By default, a **buffer** should be instantiated inside the channel during elaboration. It will be used to store the initial information contained in the TDF output port bound to the channel, before starting simulation; and the information produced by the TDF output port, during simulation. The size of this circular data buffer is determined after performing, on the TDF cluster, the DE-TDF pre-simulation analysis previously presented in Section 4.4.

- Additional **in_delay_buffers** should be also instantiated inside the channel during elaboration. They will be used to store the initial information contained in each TDF input port bound to the channel, before starting simulation. The **in_delay_buffers** size will be determined by the delay attribute value associated to each one of the TDF input ports bound to the channel.

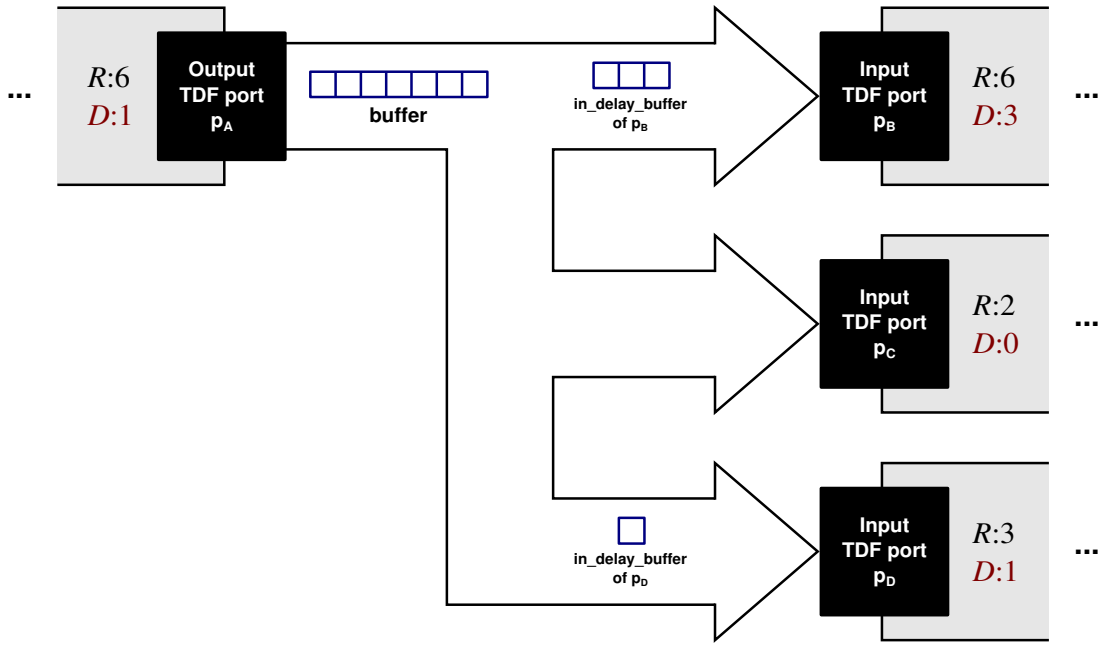

Figure 6.1: Specification of the TDF Predefined Channel.

The **buffer** and **in_delay_buffers** instantiated inside the channel should be accessed thanks to the *channel interface*, which will be defined by means of three methods: initialize(), read() and write().

1. initialize(p, val, id): is the method called from a TDF input or output port for initializing the values of ports with a delay attribute previously assigned. This method, in addition to receiving as argument the reference of the port p to be initialized, receives:

   - The initial data value val of the sample to be stored in the channel before starting simulation.
   - The index id of the sample being initialized. A sample can be indexed from zero to a value less than the delay attribute value associated to the port calling the initialize() method.

   As shown in Figure 6.2, according to the type of TDF port calling this method, different data buffers can be initialized.

   (a) When the method is called from a TDF input port p, the data value val is stored in the position id of the **in_delay_buffer** corresponding to such port p.

   (b) When the method is called from a TDF output port p, the data value val is stored in the position id of the **buffer** instantiated inside the channel.

2. read(p, id): is the method called from a TDF input or output port for reading a data value contained inside the channel. This method in addition of receiving as argument the reference of the port p, which wants to read; it receives the index id of the sample to be read. This index should be less than the rate attribute associated to the port p.

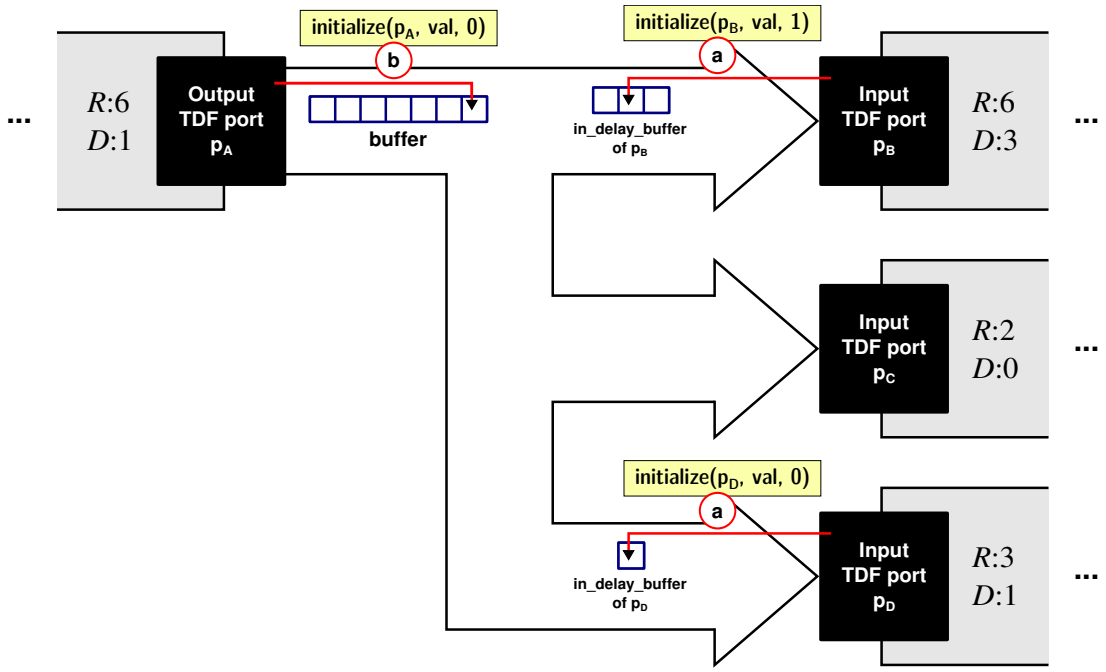

Figure 6.2: Calling initialize() Method in the TDF Predefined Channel.

According to the reading raw position $read_{\mathrm{rpos}}$ (determined by the Equation 6.1) and the type of port calling this method, different data buffers can be read. In this equation, j_{M} is the number of times that the module M has been executed (M is the module where the port p is instantiated), R_{p} is the rate associated to port p, D_{p} is the delay associated to port p, and id is the index of the sample that the port p wants to read.

$$read_{\mathrm{rpos}} = (j_{\mathsf{M}} * R_{\mathsf{p}}) - D_{\mathsf{p}} + \mathsf{id} \tag{6.1}$$

As shown in Figure 6.3, when the $read_{\mathrm{rpos}} < 0$, a *delay sample* should be read from the channel, according to the next conditions:

ⓐ If the method is called from a TDF input port p, the *delay sample* is read from the $read_{\mathrm{pos}}$ of the **in_delay_buffer** corresponding to port p. This $read_{\mathrm{pos}}$ is determined by the Equation 6.2.

ⓑ If the method is called from a TDF output port p, the *delay sample* is read from the $read_{\mathrm{pos}}$ of the **buffer** instantiated inside the channel. This $read_{\mathrm{pos}}$ is determined by the Equation 6.2.

$$read_{\mathrm{pos}} = (j_{\mathsf{M}} * R_{\mathsf{p}}) + \mathsf{id} \tag{6.2}$$

Conversely, when the $read_{\mathrm{rpos}} \geq 0$, a *sample* should be read from the channel, according to the next conditions:

ⓒ If the method is called from a TDF input port p, the *sample* is read from the $read_{\mathrm{pos}}$ of the **buffer** instantiated inside the channel. In this case, the $read_{\mathrm{pos}}$ is determined by the Equation 6.3, where B_{size} is the size of the **buffer** instantiated inside the channel.

$$read_{\mathrm{pos}} = ((j_{\mathsf{M}} * R_{\mathsf{p}}) - D_{\mathsf{p}} + \mathsf{id}) \ \% \ B_{\mathrm{size}} \tag{6.3}$$

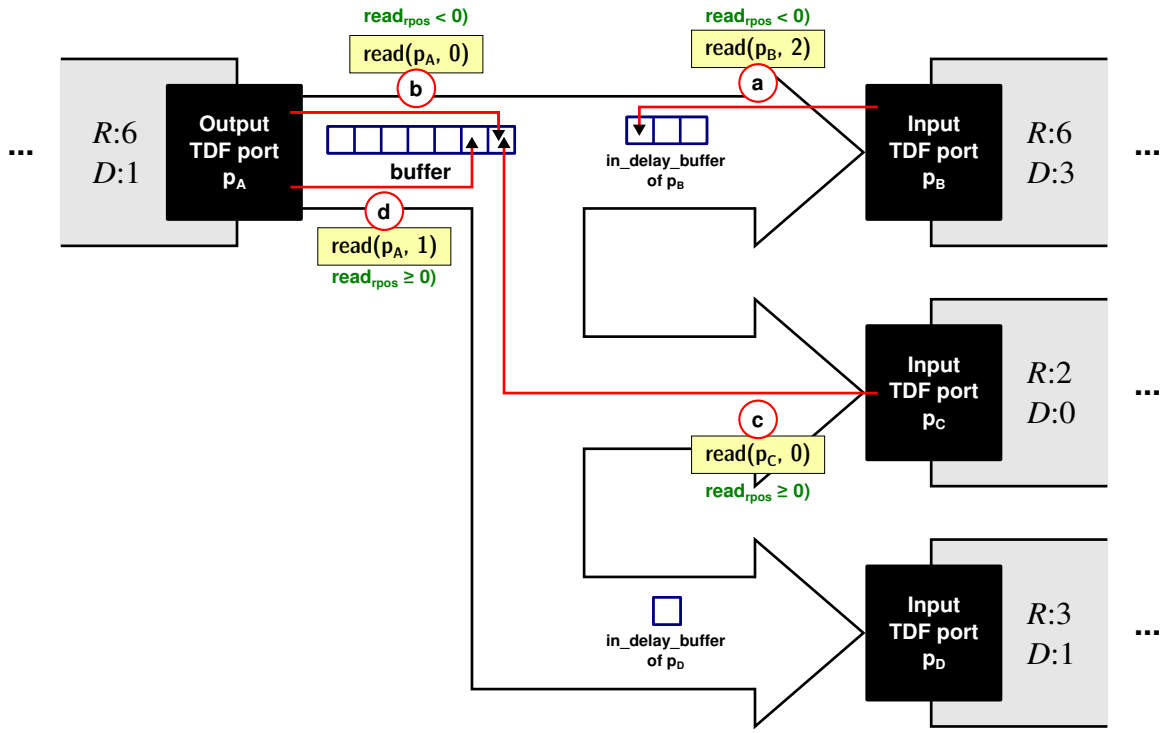

Figure 6.3: Calling read() Method in the TDF Predefined Channel.

(d) If the method is called from a TDF output port p, the *sample* is read from the $read_{pos}$ of the **buffer** instantiated inside the channel. In this case, the $read_{pos}$ is determined by the Equation 6.4.

$$read_{pos} = ((j_M * R_p) + id) \ \% \ B_{size} \tag{6.4}$$

3. write(p, val, id): is the method called from a TDF output port to write a data value in the **buffer** instantiated inside the channel. This method in addition of receiving as argument the reference of the port p, which wants to write; it receives:

   - The data value val of the sample to be written in the channel.
   - The index id of the sample to be written. This index should be less than the rate attribute associated to the port p.

   The position of the **buffer** where the sample is written, is determined by the Equation 6.5. There, j_M is the number of times that the module M has been executed (M is the module where the port p is instantiated), R_p is the rate associated to port p, D_p is the delay associated to port p, id is the index of the sample that the port p wants to write, and B_{size} is the size of the **buffer** instantiated inside the channel. An example is shown in Figure 6.4.

$$write_{pos} = ((j_M * R_p) + D_p + id) \ \% \ B_{size} \tag{6.5}$$

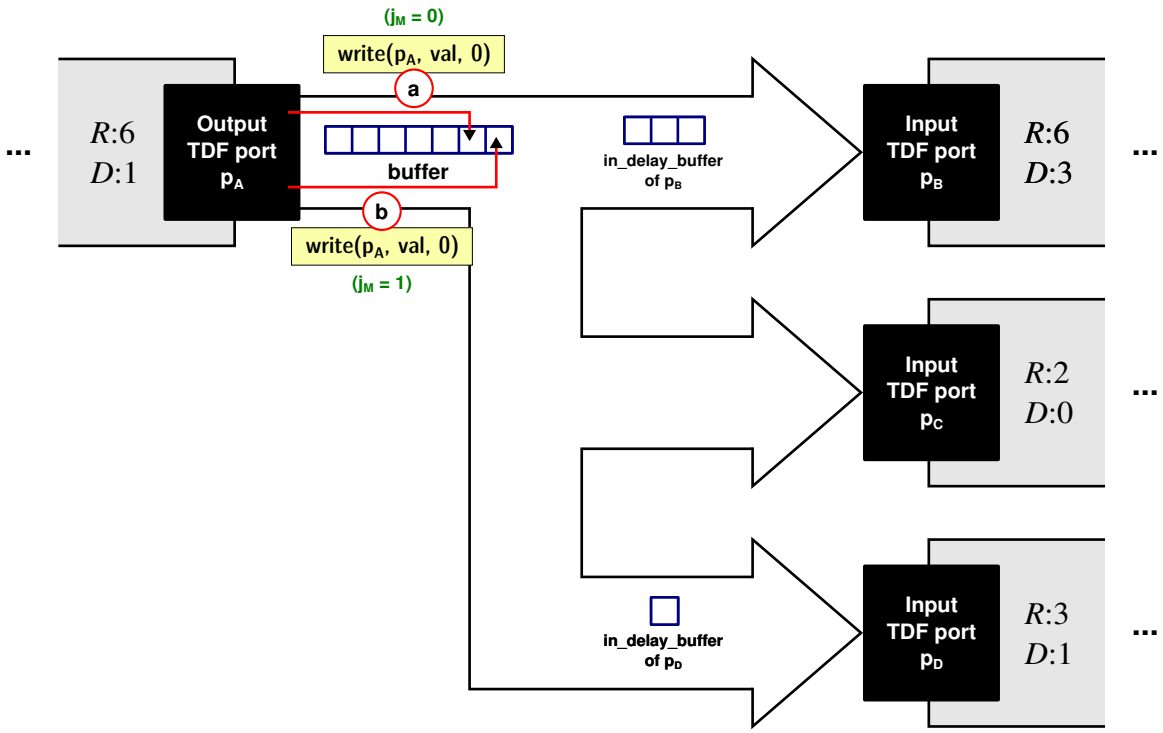

Figure 6.4: Calling write() Method in the TDF Predefined Channel.

c. Classical TDF Ports

The specification of classical TDF ports in SystemC MDVP is not the responsibility of the designer. In the TDF MoC, two types of predefined classical ports should be available to be instantiated inside TDF modules. These ports will be specified by means of two classes: sca_tdf::sca_in (for classical input TDF ports), and sca_tdf::sca_out (for classical output TDF ports). Implementation details of these classes are presented in Section 6.4.3.

On the one hand, designers should be able to initialize *classical TDF input ports* before starting simulation, and read them during simulation. It will be possible by means of the initialize() and read() methods implemented in such ports, as shown in Figure 6.5.

As a classical TDF input port **n** can access to the channel **S** to which it is bound, the methods initialize() and read() will be implemented as functions calling the initialize() and read() methods respectively implemented in the channel **S**.

On the other hand, designers should be able to initialize *classical TDF output ports* before starting simulation, read and write them during simulation. It will be possible by means of the initialize(), read() and write() methods implemented in such ports, as shown in Figure 6.5.

As a classical TDF output port **m** can access the channel **S** to which it is bound, the methods initialize(), read() and write() are implemented as functions calling the initialize(), read() and write() methods respectively implemented in the channel **S**.

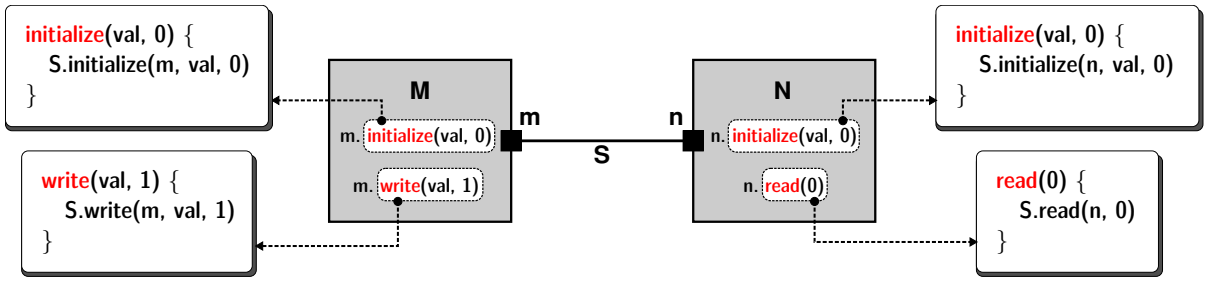

Figure 6.5: Methods Implemented in Classical TDF Ports.

6.2.3. Location of the TDF MoC inside the SystemC MDVP Architectural Model

As TDF is the first model of computation added to SystemC MDVP, it is located at the second hierarchical level of the architectural model introduced in Section 5.4.1. This means that the TDF MoC is located under the DE MoC. Therefore, it should define:

- Converter ports to handle the *data synchronization* between the TDF MoC and the DE MoC.
- A solver able to handle the *time synchronization* between the TDF MoC and the DE MoC.

a. TDF Input Converter Ports

TDF input converter ports should be responsible for handling the data synchronization from the DE MoC to the TDF MoC. To this end, we have decided that a TDF input converter port should be specified as an abstract data type, which contains circular data buffers used to temporarily store the information transmitted from a DE module to a TDF module. An example is shown in Figure 6.6.

- By default, an **in_buffer** should be instantiated inside the TDF input converter port during elaboration. It will be used to store the information read from the DE signal (associated to the input converter port) during simulation. The size of this circular buffer is determined after performing, on the TDF cluster, the DE-TDF pre-simulation analysis previously presented in Section 4.4.
- One additional **in_delay_buffer** should be also instantiated inside the TDF input converter port during elaboration. It will be used to store the initial information contained in the TDF input converter port, before starting simulation. The **in_delay_buffer** size will be determined by the delay attribute value associated to the TDF input converter port.

Besides, as shown in Figure 6.6, we have decided to handle the DE-TDF data synchronization by means of two methods: read_sc_signal() and read(). These methods are described below.

ⓐ read_sc_signal(t): is the method called by the simulator for reading at DE time t, a data value contained inside the DE channel bound to the input converter port p; and next, writing a sample with such data value in the **in_buffer** instantiated inside p.

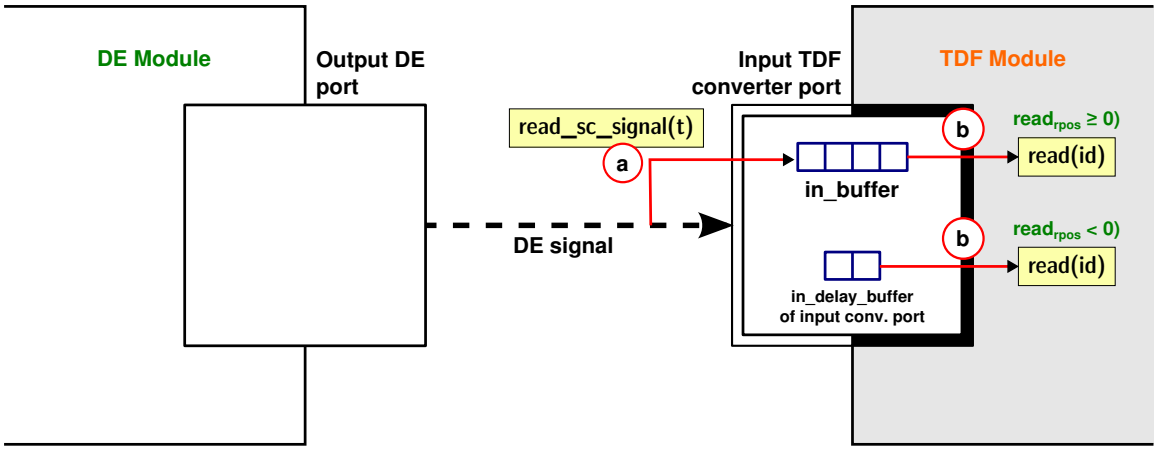

Figure 6.6: Specification of TDF Input Converter Ports.

The position of the **in_buffer** where the sample is written, is determined by the Equation 6.6. There, t the DE time at which the *DE-TDF data synchronization* is required, Tp_{p} is the time step associated to port p, and $B_{in_{\mathsf{size}}}$ is the size of the **in_buffer** instantiated inside the port p.

$$write_{\mathsf{pos}} = \left(\frac{\mathsf{t}}{Tp_{\mathsf{p}}} \right) \% \, B_{in_{\mathsf{size}}} \tag{6.6}$$

(b) read(id): is the method called by the designer for reading a sample from the TDF input converter port p. This method receives as argument the index id of the sample to be read. This index should be less than the rate attribute associated to the port p.

According to a reading raw position $read_{\mathsf{rpos}}$ (determined by the Equation 6.7), different buffers can be read . In this equation, j_{M} is the number of times that the module M has been executed (M is the module where the port p is instantiated), R_{p} is the rate associated to port p, D_{p} is the delay associated to port p, and id is the index of the sample to be read.

$$read_{\mathsf{rpos}} = (j_{\mathsf{M}} * R_{\mathsf{p}}) - D_{\mathsf{p}} + \mathsf{id} \tag{6.7}$$

As shown in Figure 6.6, when the $read_{\mathsf{rpos}} < 0$, a *delay sample* should be read from the $read_{\mathsf{pos}}$ of the **in_delay_buffer** instantiated inside p. In this case, the $read_{\mathsf{pos}}$ is determined by the Equation 6.8.

$$read_{\mathsf{pos}} = (j_{\mathsf{M}} * R_{\mathsf{p}}) + \mathsf{id} \tag{6.8}$$

Conversely, when the $read_{\mathsf{rpos}} \geq 0$, a *sample* should be read from the $read_{\mathsf{pos}}$ of the **in_buffer** instantiated inside p. In this case, the $read_{\mathsf{pos}}$ is determined by the Equation 6.9, where $B_{in_{\mathsf{size}}}$ is the size of the **in_buffer** instantiated inside the port p.

$$read_{\mathsf{pos}} = ((j_{\mathsf{M}} * R_{\mathsf{p}}) - D_{\mathsf{p}} + \mathsf{id}) \% \, B_{in_{\mathsf{size}}} \tag{6.9}$$

In addition, the TDF input converter port p should provide to the designer, a method initialize(val, id) for initializing the **in_delay_buffer** in the position id, when p has a delay attribute assigned. This method, receives as arguments:

- The initial data value val of the sample to be stored in p before starting simulation.
- The index id of the sample being initialized. A sample can be indexed from zero to a value less than the delay attribute value associated to p.

b. TDF Output Converter Ports

TDF output converter ports should be responsible for handling the data synchronization from the TDF MoC to the DE MoC. To this end, we have decided that a TDF output converter port should be specified as an abstract data type, which contains a circular data buffer, called **out_buffer**, as shown in Figure 6.7. It will be used to temporarily store the information to be written in the DE signal (bound to the output converter port) during simulation. The size of the **out_buffer** is determined after performing, on the TDF cluster, the DE-TDF pre-simulation analysis previously presented in Section 4.4.

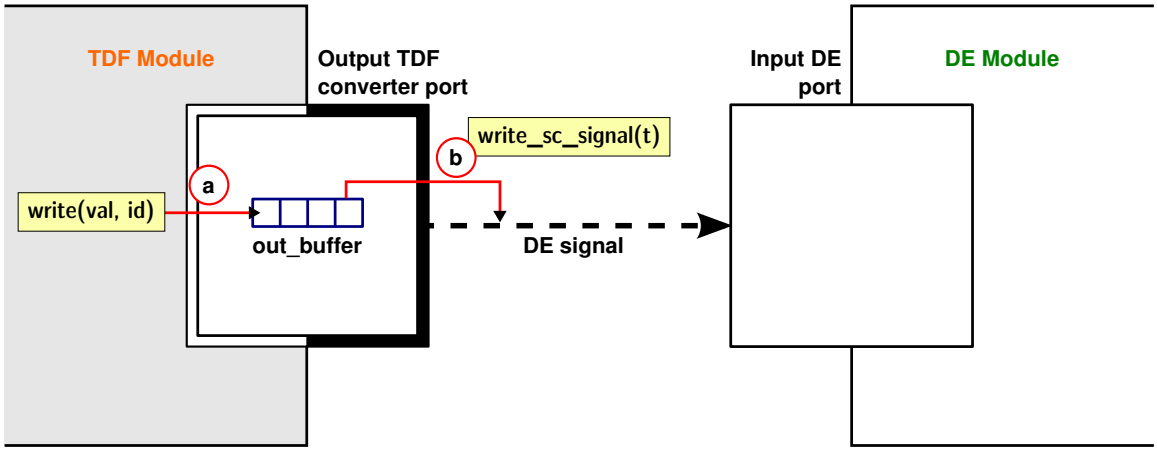

Figure 6.7: Specification of TDF Output Converter Ports.

Besides, as shown in Figure 6.7, we have decided to handle the *TDF-DE data synchronization* by means of two methods: write() and write_sc_signal(). These methods are described below.

ⓐ write(val, id): is the method called by the designer for writing a sample on the TDF output converter port p. This method receives as arguments the data value val of the sample to be stored inside the **out_buffer** of port p, and the index id of the sample to be written.

The position of the **out_buffer** where the sample is written, is determined by the Equation 6.10. There, j_M is the number of times that the module M has been executed (M is the module where the port p is instantiated), R_p is the rate associated to port p, D_p is the delay associated to port p, id is the index of the sample to be written, and $B_{out_{size}}$ is the size of the **out_buffer** instantiated inside the port p.

$$write_{pos} = ((j_M * R_p) + D_p + id) \ \% \ B_{out_{size}} \tag{6.10}$$

ⓑ write_sc_signal(t): is the method called by the simulator for reading a sample from the **out_buffer** instantiated inside p; and next, writing at DE time t, a data value on the DE channel bound to the output converter port p.

The position of the **out_buffer** where the sample is read, is determined by the Equation 6.11. There, t is the DE time at which the TDF-DE data synchronization is required, Tp_{p} is the time step associated to port p, and $B_{out_{\mathsf{size}}}$ is the size of the **out_buffer** instantiated inside the port p.

$$write_{\mathsf{pos}} = \left(\frac{\mathsf{t}}{Tp_{\mathsf{p}}} \right) \% \, B_{out_{\mathsf{size}}} \tag{6.11}$$

In addition, the TDF output converter port p should provide to the designer, a method initialize(val, id) for initializing the **out_buffer** in the position id, when p has a delay attribute assigned. This method, receives as arguments:

- The initial data value val of the sample to be stored in p before starting simulation.
- The index id of the sample being initialized. A sample can be indexed from zero to a value less than the delay attribute value associated to p.

c. DE-TDF Solver

The *DE-TDF solver* should be responsible for handling the time synchronization between the DE and TDF MoCs. To this end, it should implement the abstract methods defined in the *DE MoC interface*. As defined in Section 5.4.2, these are the abstract methods elaborate() and simulate(). In the TDF MoC:

- The elaborate() method implements the *TDF elaboration phase*, presented in Section 6.3.1.
- The simulate() method implements the *TDF simulation phase*, presented in Section 6.3.2.

6.3. TDF Elaboration and Simulation Phases in SystemC MDVP

Based on the semantics defined by the SystemC MDVP kernel, we extend the elaboration and simulation phases as shown in Figure 6.8. These phases will be performed by each *DE-TDF solver* instantiated in a model.

On the one hand, during the *TDF elaboration*, we add methods to perform the attribute settings on each one of the modules or ports encapsulated inside the TDF cluster, on which the DE-TDF solver is instantiated; calculate and propagate the time step attributes between such modules and ports; and check the TDF cluster's computability.

On the other hand, during the *TDF simulation*, we add methods to perform the initialization and the registration of the processing of TDF modules belonging to the TDF cluster, on which the DE-TDF solver is instantiated.

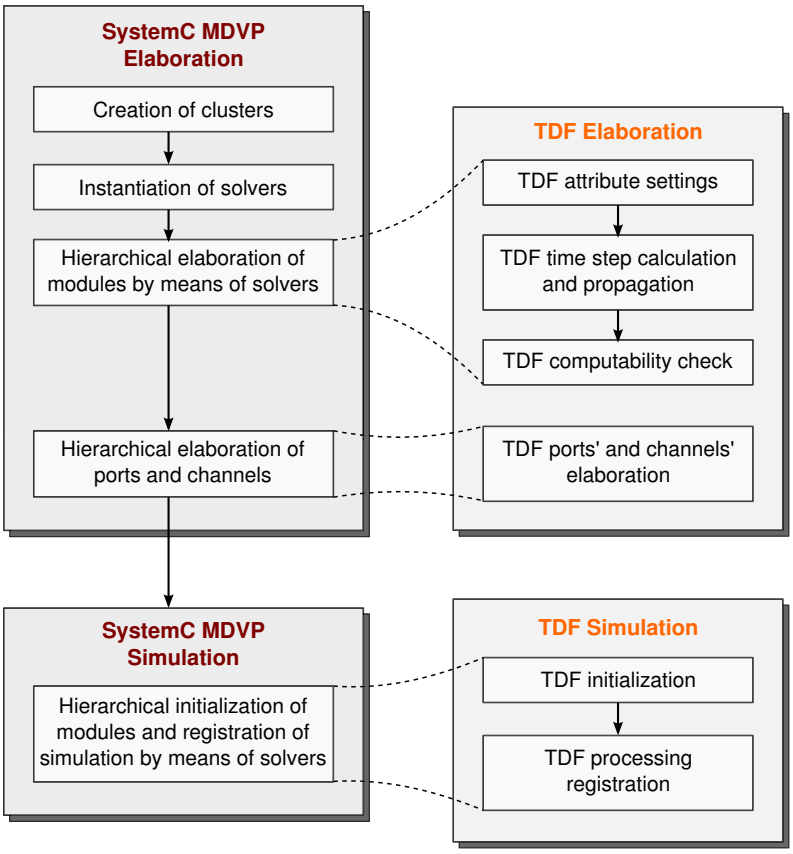

Figure 6.8: TDF Elaboration and Simulation Phases.

6.3.1. TDF Elaboration Phase

a. TDF Attribute Settings

When the designer writes a TDF model, he can specify by means of the set_attributes() method, offered by the *TDF MoC interface*, the attributes to be assigned to each module or port there instantiated.

When the elaboration phase begins, the first stage performed is the *TDF attribute settings*. In this stage, using the hierarchy of solvers constructed by SystemC MDVP (see Section 5.5.1.b), the set_attributes() function is called on each TDF module instantiated by the designer, and on each solver instantiated by the simulator on the clusters which want to interact with TDF. An example is shown in Figure 6.9.

In the case of TDF modules, the TDF attribute settings corresponds to the execution of the implementation defined by the designer inside the set_attributes() function, where for example, the designer can assign a time step to a module (by means of the set_timestep() function) or assign a time step, rate or delay attribute to a port (by means of the set_timestep(), set_rate() or set_delay() functions).

In the case of solvers, which want to interact with TDF, the set_attributes() corresponds to the execution of the specific elaboration phases of the MoC where such solver is defined.

At the end of this stage, the simulator verifies that at least one component per TDF cluster has a time step attribute assigned. It is a required condition to continue the TDF elaboration.

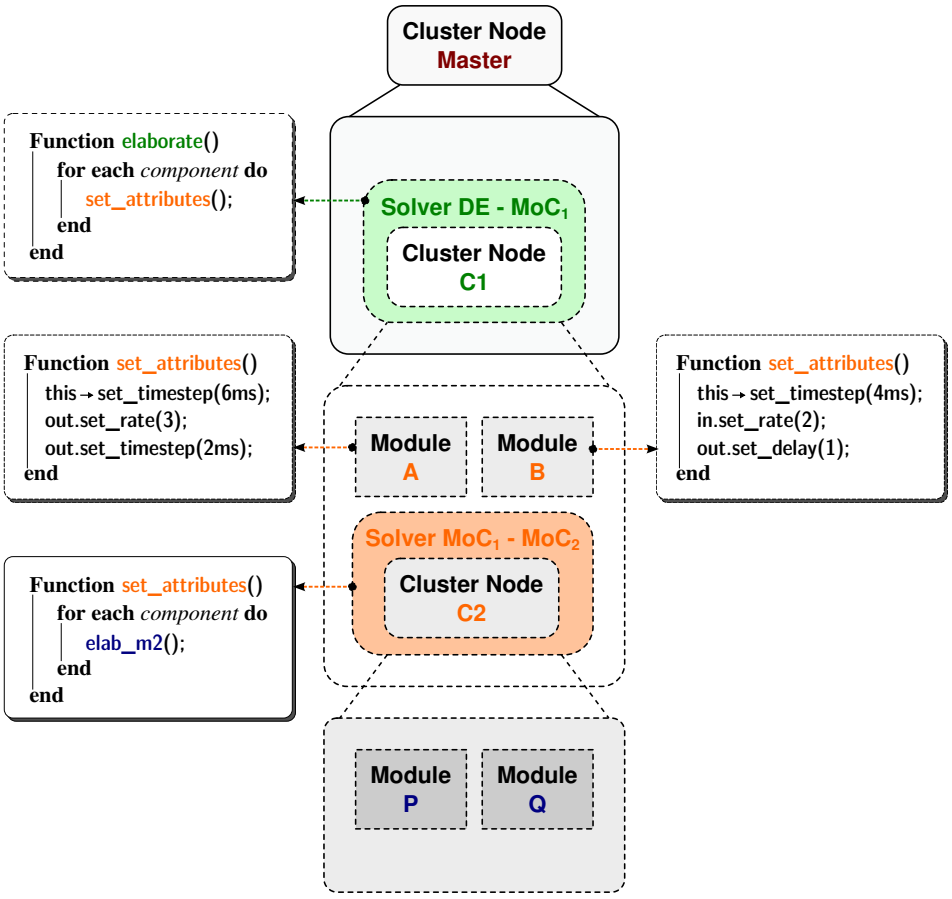

Figure 6.9: Function set_attributes() in the Hierarchy of Solvers.

b. TDF Time Step Calculation and Propagation

When at least one time step has been assigned inside a cluster, it should be propagated to the remaining modules or ports (belonging to the same cluster) that do not contain it. Thanks to the methods offered by SystemC MDVP, we can traverse the model in depth to assign or verify, according to the rules presented below, the time step value associated to each module or port instantiated inside the TDF cluster. When the time step has been assigned in a module, the propagation is performed as introduces (b.1). Conversely, when the time step has been assigned in a port, the propagation is performed as introduces (b.2).

(b.1) **Propagate the time step from a module M**: when the time step is assigned to a module M, this time step should be propagated to each port m belonging to M, according to the Equation 6.12. There, Tp_{m} is the port time step to be propagated, Tm_{M} is the module time step, and R_{m} is the rate attribute assigned to port m.

$$Tp_{\mathsf{m}} = \frac{Tm_{\mathsf{M}}}{R_{\mathsf{m}}} \tag{6.12}$$

Then, starting from the port where the time step was propagated, a new propagation should be performed.

b.2 **Propagate the time step from a port m**: when the time step is assigned to a port m, this time step should be propagated:

- To the module M, which contains m, according to the Equation 6.13. Then, starting from the module where the time step was propagated, a new propagation should be performed.

$$Tm_{\mathsf{M}} = Tp_{\mathsf{m}} * R_{\mathsf{m}} \tag{6.13}$$

- To each one of the ports n connected to m, according to the Equation C.5, as long as the port m is not a TDF conversion port. Then, starting from the port where the time step was propagated, a new propagation should be performed.

$$Tp_{\mathsf{n}} = Tp_{\mathsf{m}} \tag{6.14}$$

The propagation of time step is performed until each module or port in the model has a time step assigned. An example of propagation is shown in Figure 6.10.

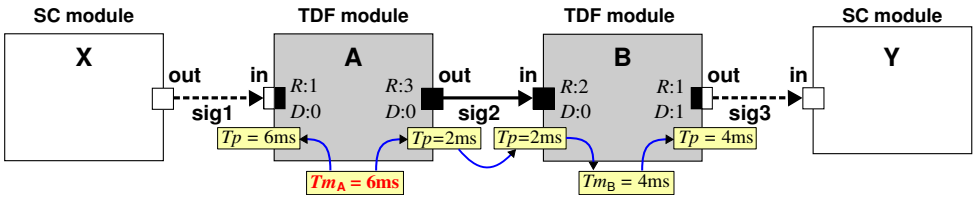

Figure 6.10: Example of Time Step Calculation and Propagation in a TDF Cluster.

When multiple time steps are assigned in a same TDF cluster, during this stage, the simulator verifies that such time steps are cosistent according to Equations 6.12 – C.5.

c. TDF Computability Check

In order to verify that a TDF cluster is computable and then, determine the schedule to be used for the execution of such cluster, in this stage we perform two analysis phases. First, an analysis phase, based on the Synchronous Data Flow (SDF) formalism [35], to verify the rate consistencies in the cluster, and calculate the number of times (q) that each module should be executed within a cluster period ($Tcls$). Second, the analysis presented in Chapter 4, to detect and propose solutions for the causality issues, which can arise in TDF models interacting with the DE domain; and determine the TDF cluster schedule, which contains not only the order in which the TDF modules should be executed, but also the order of its interactions with the DE domain.

Considering the example shown in Figure 6.10, we explain how the simulator applies the SDF formalism to verify the rate consistencies within the TDF cluster. Knowing that the precondition in the TDF standard for a correct data synchronization is that the value read from a converter port should be available at the first delta cycle of the corresponding time point in the DE domain [28], each TDF cluster can be isolated to be initially analyzed without considering its interactions with the DE domain. In the example, as shown in Figure 6.11, the TDF cluster is isolated from full model.

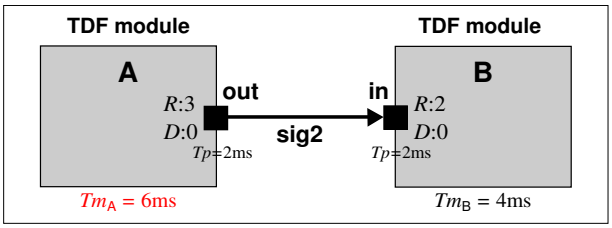

Figure 6.11: Isolated TDF Cluster Composed of 2 TDF Modules and 1 TDF Signal.

Following the SDF formalism, it is possible to build a SDF graph from a TDF cluster, as shown in Figure 6.12. TDF modules are represented as nodes, TDF signals are represented as edges, output rates are represented as the number of samples produced to an edge, and input rates are represented as the number of samples consumed from an edge.

Figure 6.12: SDF Graph Representing the Isolated TDF Cluster.

Analyzing this SDF graph, a topology matrix $\Gamma_{i,j}$ can be calculated, where each (i, j)th entry is the amount of data produced by a node j on an arc i. The number of columns on the matrix corresponds to the number of existing nodes in the SDF graph, and the number of rows corresponds to the number of edges.

Rate consistencies can be determined calculating the rank of the matrix $\Gamma_{i,j}$, which should be equal to $N-1$ (being N the number of existing nodes in the SDF graph). It is a necessary condition to ensure the existence of a valid schedule. This means that a valid schedule cannot be found when the condition is not fulfilled. For the example, the condition is verified in Equation 6.15.

$$\Gamma_{i,j} = \begin{bmatrix} 3 & -2 \end{bmatrix}$$

$$Rank\left(\Gamma_{i,j}\right) = 1 = N - 1 \tag{6.15}$$

After rate verifications, the number of executions $q_{j,1}$ of each TDF module during a cluster period can be determined using the Equation 6.16. This corresponds to find a solution to the system of equations proposed. As shown in the example: module **A** will be executed twice and module **B** will be executed three times.

$$\Gamma_{i,j} \cdot q_{j,1} = 0 \tag{6.16}$$

$$\begin{bmatrix} 3 & -2 \end{bmatrix} \cdot \begin{bmatrix} q_A \\ q_B \end{bmatrix} = 0$$

$$\begin{bmatrix} q_A \\ q_B \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Knowing the number of executions of each TDF module (q) and its time step (Tm), the period of the TDF cluster ($Tcls$) can be found using the Equation 6.17. The cluster period in the example is 12 ms.

$$Tcls = Tm_{\mathrm{j}} \cdot q_{\mathrm{j}} \tag{6.17}$$

$$Tcls = Tm_{\mathrm{A}} \cdot q_{\mathrm{A}} = Tm_{\mathrm{B}} \cdot q_{\mathrm{B}}$$
$$Tcls = 6\,\mathrm{ms} \cdot 2 = 4\,\mathrm{ms} \cdot 3$$
$$Tcls = 12\,\mathrm{ms}$$

Once the analysis based on the SDF formalism is accomplished, the simulator considers again the model shown in Figure 6.10, and creates its CPN equivalent model following the rules proposed in Section 4.3.2. Then, it performs the analysis proposed in Section 4.4. For the example, the results of this analysis were summarized in Table 4.1.

d. TDF Elaboration of Ports and Channels

As introduced in Section 5.5.1.d, the elaboration of ports and channels in a SystemC MDVP model is performed by means of the execution of the elaborate() functions implemented in the MoC. In the case of TDF, only the channels and converter ports need to be elaborated, then:

- A function elaborate() is implemented in the predefined TDF channel, in order to create the circular data buffers used to temporarily store the information transmitted through the channel. These are the **buffers** and **in_delay_buffers** presented in Section 6.2.2.b.
- A function elaborate() is implemented in the TDF input converter port, in order to create the circular data buffers used to store the information transmitted from a DE module to a TDF module. These are the **in_buffer** and **in_delay_buffer** presented in Section 6.2.3.a
- A function elaborate() is implemented in the TDF output converter port, in order to create the circular data buffer used to store the information to be transmitted from a TDF module to a DE module. This is the **out_buffer** presented in Section 6.2.3.b.

Note: the size of buffers to be instantiated is identified by the simulator during the analysis of the TDF cluster in form of equivalent CPN model.

6.3.2. TDF Simulation Phase

a. TDF Initialization

When the designer writes a TDF model, he can specify by means of the initialize() function offered by the *TDF MoC interface*, the initial values of the samples that will be stored in channels or TDF converter ports before starting simulation.

Similar than the phase of *TDF attribute settings*, in this stage, using the hierarchy of solvers constructed by SystemC MDVP (see Section 5.5.1.b), the initialize() function is called on each TDF module instantiated by the designer, and on each solver instantiated by the simulator on the clusters which want to interact with TDF. At the end of this stage, the TDF model is ready to be simulated.

b. TDF Processing Registration

In this stage, the registration of the TDF module's execution is performed by means of the sc_spawn() method provided by the SystemC standard. This method allows the simulator to create and register a dynamic process [24] in the SystemC DE kernel after the sc_start() has been called.

In the TDF MoC, the process to be registered in the SystemC DE kernel is the responsible of executing the schedule determined during the analysis of the TDF cluster in form of equivalent CPN model. The algorithm of this process is summarized in Listing 6.1.

As the process is registered in the SystemC DE kernel, it will be automatically called by the SystemC scheduler. It is considered as a simulation thread, which can be suspended by means of wait() statements.

```
1  void execute_schedule() {
2
3     sca_core::sca_time last_time;
4     sca_core::sca_time current_time;
5
6     while(1) {
7        last_time = sc_core::SC_ZERO_TIME;
8        for each element of the schedule {
9           current_time = element.time();
10          if (current_time != last_time) {
11             sc_core::wait(current_time − last_time);
12             element.execute(sc_core::sc_time_stamp());
13             last_time = current_time;
14          } else {
15             element.execute(sc_core::sc_time_stamp());
16          }
17       }
18       sc_core::wait(Tcls − current_time);
19    }
20
21 };
```

Listing 6.1: Algorithm of the Process Registered in the DE Kernel for Executing the Schedule of a TDF Cluster.

This process implements an infinite loop, which traverses the schedule and executes each one of its elements (Listing 6.1, lines 6–19). When the time associated to the schedule's element to be executed (current_time) is different from the time of the last schedule's element executed (last_time), a wait() statement is registered in the DE kernel. After this wait(), the schedule's element is executed (Listing 6.1, lines 10–14). Conversely, when the time associated to the schedule's element to be executed (current_time) is equal to the time of the last schedule's element executed (last_time), then the schedule's element is immediately executed (Listing 6.1, lines 14–16). Once the schedule has been traversed, another wait() statement is automatically registered to suspend the process until the next TDF time period, where the schedule will be re-executed (Listing 6.1, line 18).

The method execute(), associated to each schedule's element, is able to identify the function to be performed during simulation:

- If the schedule's element corresponds to a *TDF module*, it executes the processing() method associated to such TDF module.

- If the schedule's element corresponds to a *DE-TDF synchronization operation*, which reads a DE signal, it executes the read_sc_signal() method associated to the input converter port bounded to the same DE signal.

- If the schedule's element corresponds to a *TDF-DE synchronization operation*, which writes a DE signal, it executes the write_sc_signal() method associated to the output converter port bounded to same DE signal.

6.4. Overview of the TDF MoC Implementation

This section describes how the TDF MoC, introduced in the previous sections, is implemented following the methodology presented in Section 5.7.

The hierarchy of classes defined for the TDF MoC is shown in Figure 6.13. These classes directly inherit from the SystemC MDVP kernel classes.

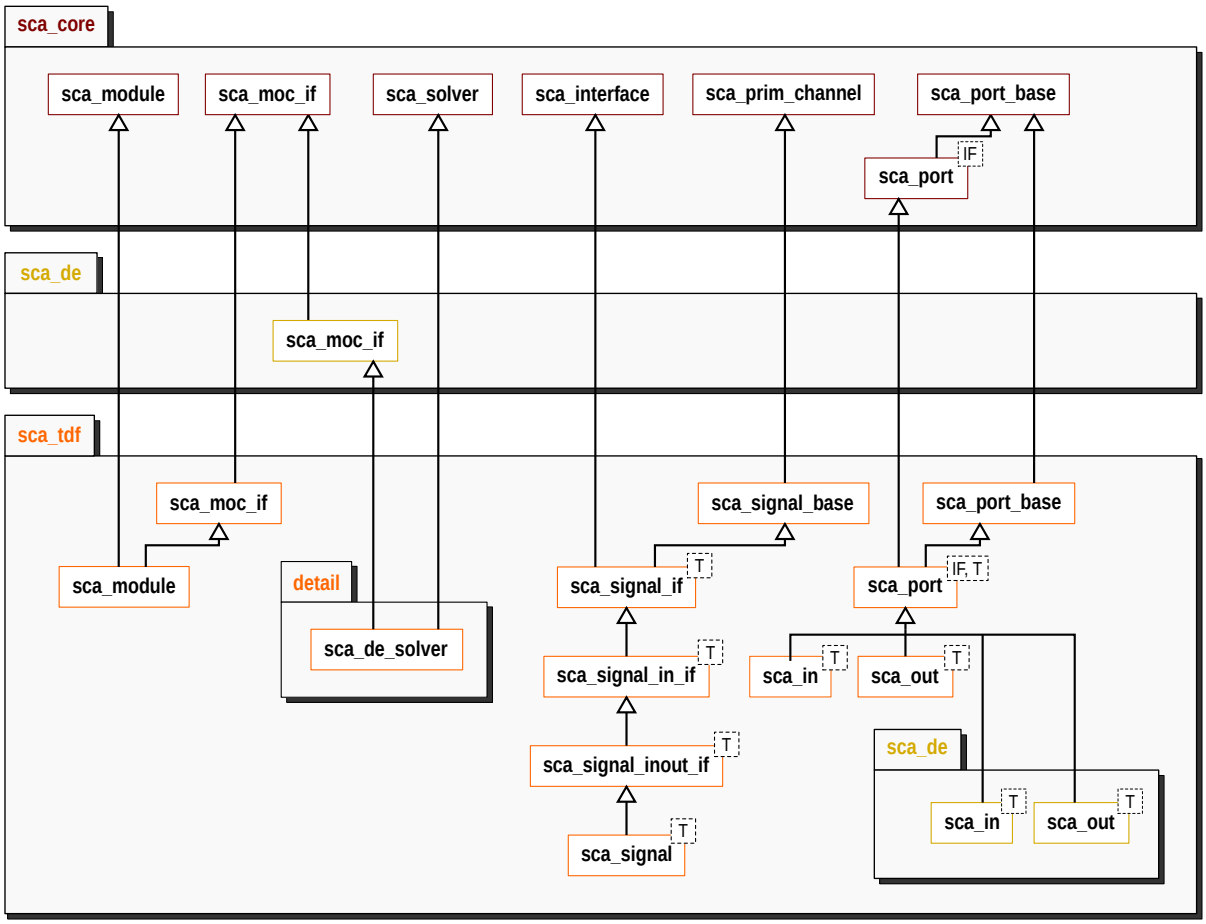

Figure 6.13: Overview of the TDF MoC Classes.

6.4.1. Implementation of the TDF Module

Following the methodology proposed in Section 5.7.1, we create two TDF classes for implementing the TDF module. These classes are shown in Figure 6.14.

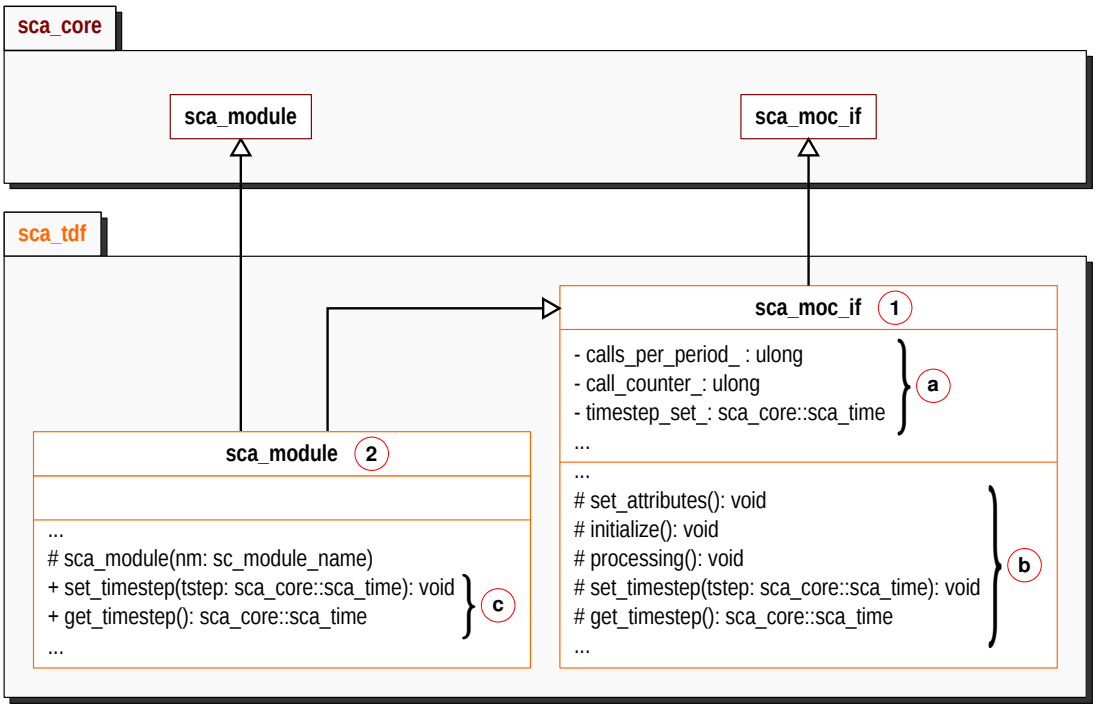

Figure 6.14: Overview of the TDF MoC Interface and Module Classes.

- The sca_tdf::sca_moc_if (1 in Figure 6.14) is the class created to handle the components of TDF clusters (TDF modules, and solvers which want to interact with TDF). This class defines the attributes of TDF modules (a in Figure 6.14), and the abstract methods integrating the *TDF MoC interface* (b in Figure 6.14). These abstract methods are used during the TDF elaboration and simulation phases.

- The sca_tdf::sca_module (2 in Figure 6.14) is the class provided to the designer to be inherited in a model, in order to represent TDF blocks with particular behaviors. It inherits the attributes and methods from the sca_tdf::sca_moc_if class, and it provides to the designer the methods (c in Figure 6.14) to assign (during elaboration) and access (during simulation) the TDF module timestep attribute.

6.4.2. Implementation of the Predefined TDF Channel

Following the methodology proposed in Section 5.7.2, we create several TDF classes for implementing the predefined TDF channel. These classes are shown in Figure 6.15.

- The sca_tdf::sca_signal_base (1 in Figure 6.15) is the class created to handle TDF predefined channels. It implements the methods:

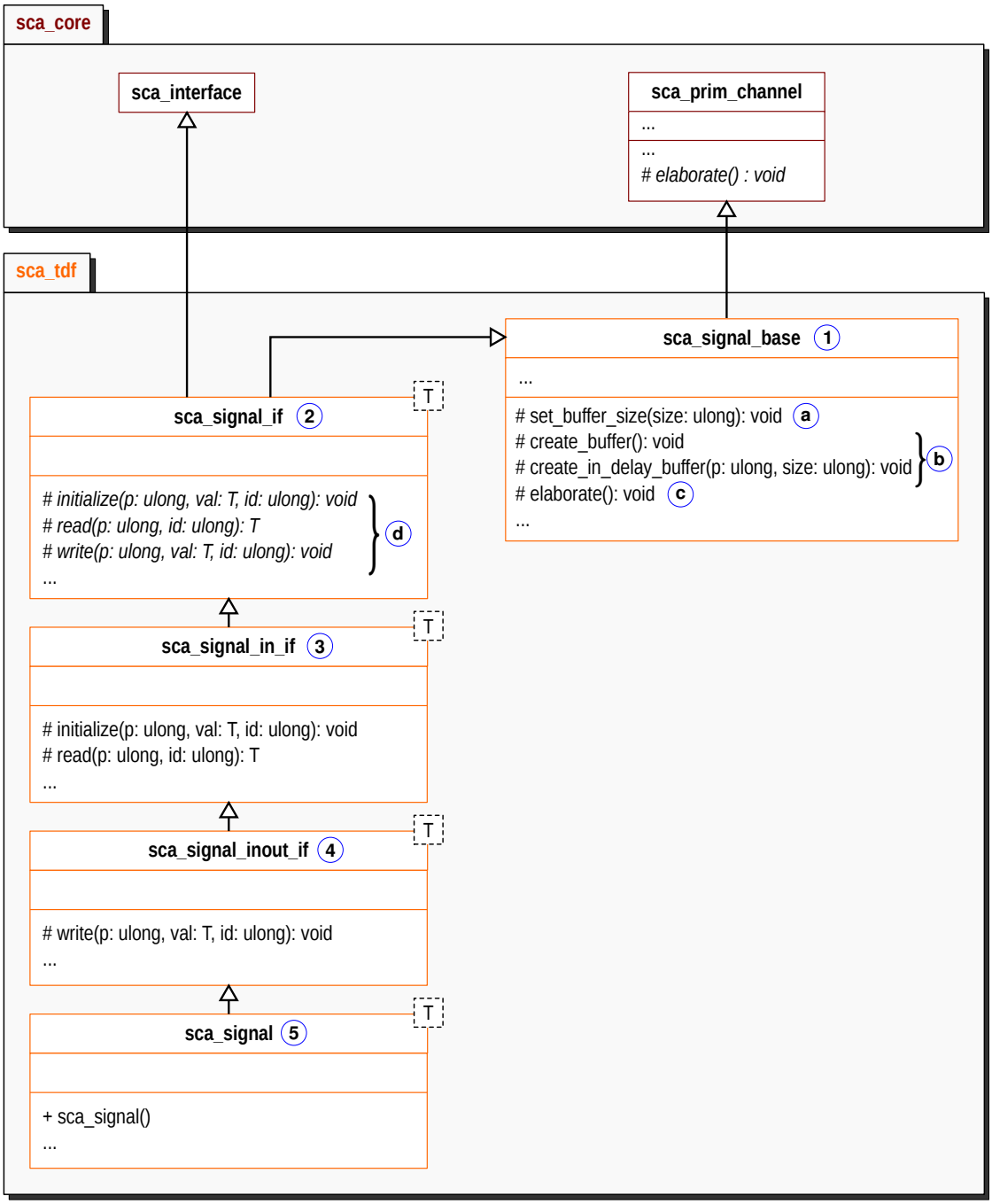

Figure 6.15: Overview of the TDF Channel Classes.

- set_buffer_size() (ⓐ in Figure 6.15), which sets the size required to instantiate the buffer in the channel. This buffer is used to store the initial information contained in the TDF output port bound to the channel, before starting simulation; and the information produced by the TDF output port, during simulation.
- create_buffer() and create_in_delay_buffer() (ⓑ in Figure 6.15), which instantiate the buffer and in_delay_buffers in the channel, as previously introduced in Section 6.2.2.b.
- elaborate() (ⓒ in Figure 6.15), which calls the create_buffer() method; and depending on the delay attributes associated to each one of the TDF input ports bound to the channel, calls the create_in_delay_buffer() method.

- The sca_tdf::sca_signal_if<T> (2 in Figure 6.15) is the class defining the interface offered by the predefined TDF channel (d in Figure 6.15).
- The sca_tdf::sca_signal_in_if<T> (3 in Figure 6.15) is the class implementing the interface to be respected by the TDF input ports. This interface is composed by the methods initialize() and read() previously described in Section 6.2.2.b.
- The sca_tdf::sca_signal_inout_if<T> (4 in Figure 6.15) is the class implementing the interface to be respected by the TDF output ports. This interface, besides inherits the methods from the sca_tdf::sca_signal_in_if<T> class, implements the method write(), also described in Section 6.2.2.b.
- The sca_tdf::sca_signal<T> (5 in Figure 6.15) is the class provided to the designer to directly instantiate the predefined channel in its model.

6.4.3. Implementation of the Predefined TDF Ports

Following the methodology proposed in Section 5.7.3, we create several TDF classes for implementing the predefined TDF ports. These classes are shown in Figure 6.16.

- The sca_tdf::sca_port_base (1 in Figure 6.16) is the class created to handle the TDF ports. This class defines the attributes of TDF ports (a in Figure 6.16), and implements the methods (b in Figure 6.16) to set and get such TDF port attributes.
- The sca_tdf::sca_port<IF,T> (2 in Figure 6.16) is the class defining and implementing the methods initialize(), read() and write() (c in Figure 6.16), which can be performed on TDF ports. These methods were previously introduced in Section 6.2.2.c.
- The sca_tdf::sca_in<T> (3 in Figure 6.16) is the class implementing the IF=sca_tdf::sca_signal_in_if interface. It provides to the designer the predefined *classical TDF input port*, which makes available:
  - The methods inherited from the sca_tdf::sca_port_base class (d in Figure 6.16), to set and get port attributes.
  - The methods initialize() and read() (e in Figure 6.16), inherited from the sca_tdf::sca_port class.
- The sca_tdf::sca_out<T> (4 in Figure 6.16) is the class implementing the IF=sca_tdf::sca_signal_inout_if interface. It provides to the designer the predefined *classical TDF output port*, which makes available:
  - The methods inherited from the sca_tdf::sca_port_base class (f in Figure 6.16), to set and get port attributes.
  - The methods initialize() and write() (g in Figure 6.16), inherited from the sca_tdf::sca_port class.
- The sca_tdf::sca_de::sca_in<T> (5 in Figure 6.16) is the class implementing the SystemC interface IF=sc_core::sc_signal_in_if. It provides to the designer the predefined *TDF input converter port*, which implements the methods:

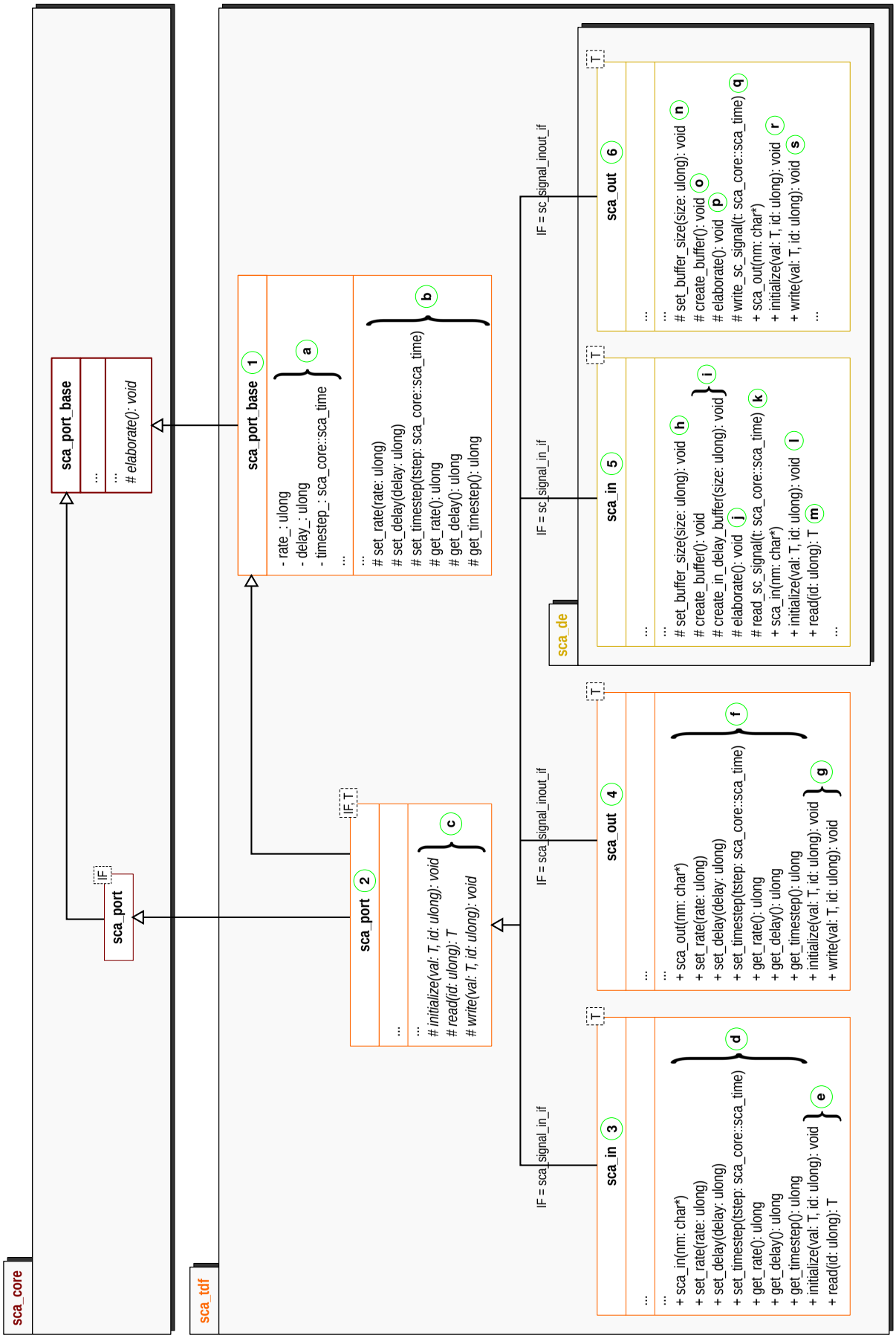

Figure 6.16: Overview of the TDF Port Classes.

  - set_buffer_size() (h in Figure 6.16), which sets the size required to instantiate the in_buffer in the TDF input converter port. This in_buffer is used to store the information read from the DE signal (bound to the TDF input converter port), during simulation.
  - create_buffer() and create_in_delay_buffer() (i in Figure 6.16), which instantiate the in_buffer and in_delay_buffer in the TDF input converter port, as previously introduced in Section 6.2.3.a.
  - elaborate() (j in Figure 6.16), which calls the create_buffer() method; and depending on the delay attributes associated to the TDF input converter port, calls the create_in_delay_buffer() method.
  - read_sc_signal() (k in Figure 6.16), which is called by the simulator to read the DE signal bound to the input converter port; and store in such TDF input converter port, a sample with the information read. This method was introduced in Section 6.2.3.a.

  In addition, this class makes available to the designer:

  - The method initialize() (l in Figure 6.16), which can be called by the designer, to store initial values in TDF input converter ports with assigned delay attributes.
  - The method read() (m in Figure 6.16), which can be called by the designer, inside the context of a processing() function, to read the TDF sample contained in the TDF input converter port, and provide this sample to the module where such port is instantiated.
  - The methods inherited from the sca_tdf::sca_port_base class (b in Figure 6.16), to set and get port attributes.

- The sca_tdf::sca_de::sca_out<T> (6 in Figure 6.16) is the class implementing the SystemC interface IF=sc_core::sc_signal_inout_if. It provides to the designer the predefined *TDF output converter port*, which implements the methods:

  - set_buffer_size() (n in Figure 6.16), which sets the size required to instantiate the out_buffer in the TDF output converter port. This out_buffer is used to store the information to be written in the DE signal (bound to the output converter port), during simulation.
  - create_buffer() (o in Figure 6.16), which instantiates the out_buffer in the TDF output converter port, as previously introduced in Section 6.2.3.b.
  - elaborate() (p in Figure 6.16), which calls the create_buffer() method.
  - write_sc_signal() (q in Figure 6.16), which is called by the simulator to write in the DE signal, bound to the output converter port, the information contained in such port. This method was introduced in Section 6.2.3.b.

  In addition, this class makes available to the designer:

  - The method initialize() (r in Figure 6.16), which can be called by the designer, to store initial values in TDF output converter ports with assigned delay attributes.
  - The method write() (s in Figure 6.16), which can be called by the designer, inside the context of a processing() function, to write in a TDF output converter port the sample generated by the module where such port is instantiated.

– The methods inherited from the sca_tdf::sca_port_base class (ⓑ in Figure 6.16), to set and get port attributes.

6.4.4. Implementation of the DE-TDF Solver

Following the methodology proposed in Section 5.7.4, we create one TDF class for implementing the DE-TDF solver responsible for executing the elaboration and simulation phases in TDF clusters. This class is shown in Figure 6.17.

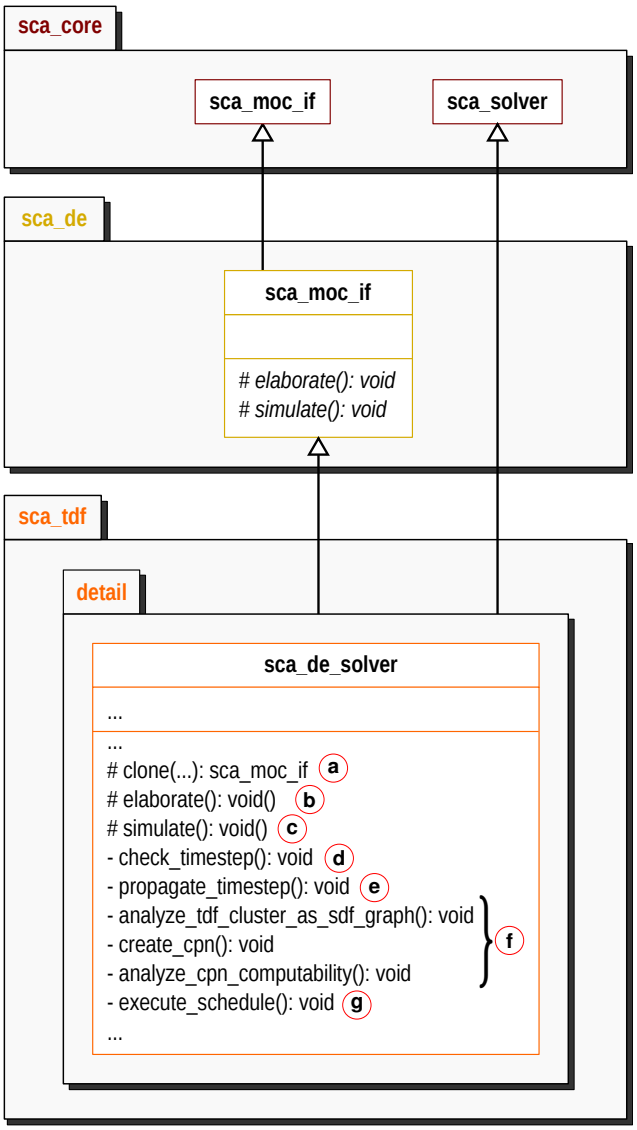

Figure 6.17: Overview of the DE-TDF Solver Class.

- The method clone() (ⓐ in Figure 6.17), returns a new instance of the DE-TDF solver. This method will be called during the phase of instantiation of solvers presented in Section 5.5.1.b.
- The method elaborate() (ⓑ in Figure 6.17), performs the TDF elaboration phase presented in Section 6.3.1. This method internally calls:
  – The set_attributes() method implemented by each module in the TDF cluster.

  - The check_timestep() method (ⓓ in Figure 6.17), which verifies that at least one TDF module or port in a TDF cluster has a time step assigned.
  - The propagate_timestep() method (ⓔ in Figure 6.17), which propagates the time step inside the TDF cluster, according to the rules presented in Section 6.3.1.b.
  - The analyze_tdf_cluster_as_sdf_graph(), create_cpn() and analyze_cpn_computability methods (ⓕ in Figure 6.17), which performs the TDF computability check stage, presented in Section 6.3.1.c.

- The method simulate() (ⓒ in Figure 6.17), performs the TDF simulation phase presented in Section 6.3.2. It internally calls:
  - The initialize() method implemented by each module in the TDF cluster.
  - The sc_core::sc_spawn() method, which registers in the SystemC DE simulation kernel, the execute_schedule() method (ⓖ in Figure 6.17). It will be responsible for executing the cluster's schedule determined during the TDF computability analysis phase, as shown in Section 6.3.2.b.

6.5. Execution of a Basic TDF Example

In order to demonstrate the advantages of the TDF MoC implemented in SystemC MDVP, we simulate, using the SystemC-AMS proof-of-concept and the SystemC MDVP simulator prototype, the example previously shown in Figure 6.10, where the delay attributes are set to zero. Results of both simulations, shown in Figures 6.18 and 6.19, are compared below. We can observe that both simulations are interrupted due to the existence of a synchronization issue in the output converter port of module **B**, but they are not interrupted at the same time.

On the one hand, in SystemC-AMS, the execution of the set_attributes() function implemented inside each TDF module is performed during the elaboration phase ①. Then, the simulation begins:

② The execution of the initialize() function implemented inside each TDF module is performed.

③ The TDF module **A** is activated. It reads one sample from the DE signal **sig1**, through the TDF input converter port **A.in**; and writes three samples on the TDF signal **sig2**, through the TDF output port **A.out**.

④ The TDF module **B** is activated. It reads two samples from the TDF signal **sig2**, through the TDF input port **B.in**; and writes one sample on the DE signal **sig3**, through the TDF output converter port **B.out**.

⑤ The TDF module **A** is activated again. It reads one sample from the DE signal **sig1**, through the TDF input converter port **A.in**; and writes three samples on the TDF signal **sig2**, through the TDF output port **A.out**.

⑥ The TDF module **B** is activated again. It reads two samples from the TDF signal **sig2**, through the TDF input port **B.in**; and detects a *synchronization issue*, which prevents the writing of a sample on the DE signal **sig3**.

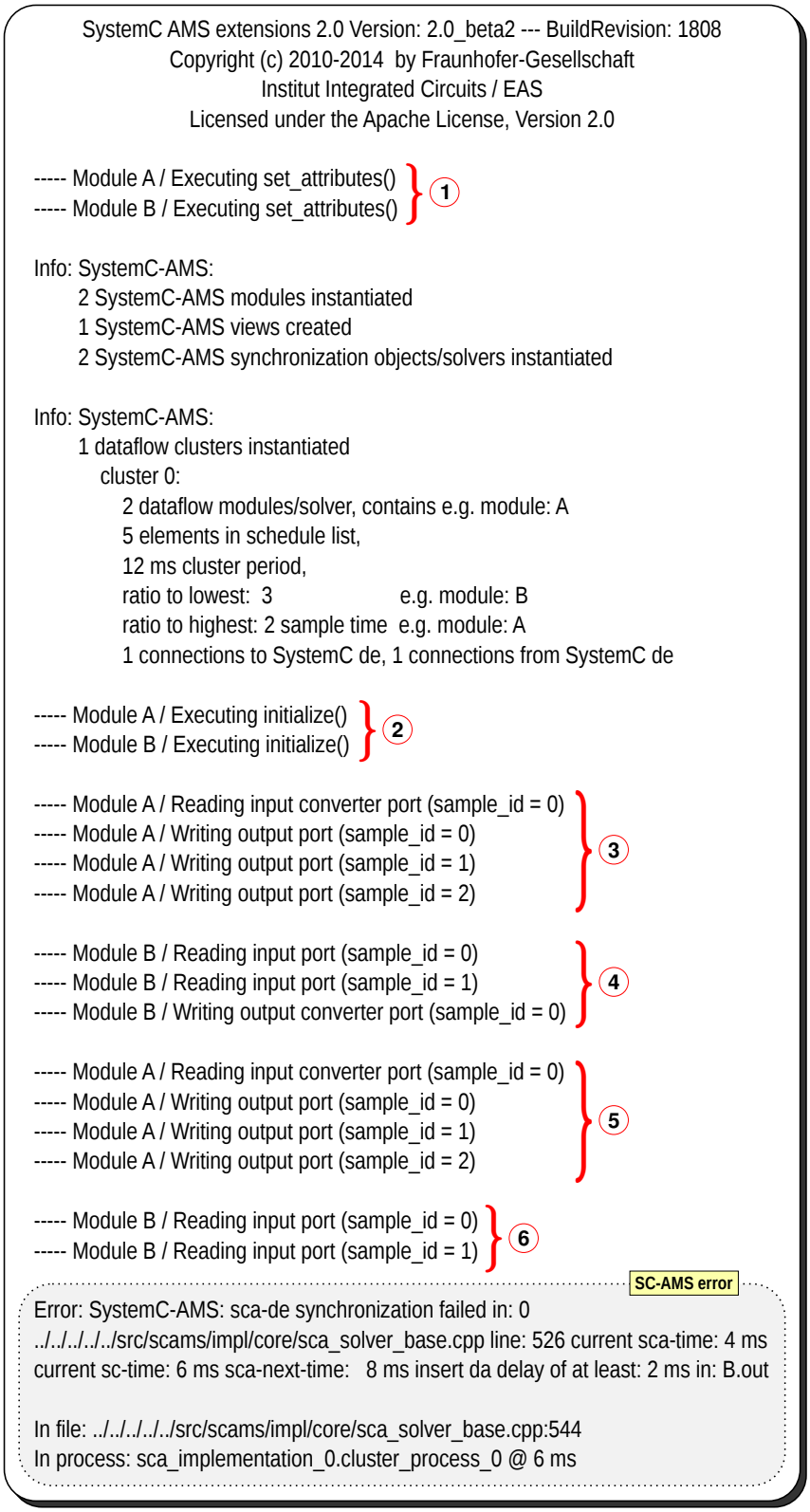

Figure 6.18: Execution of the TDF Model shown in Figure 6.10, using SystemC-AMS.

The SystemC-AMS simulation trace corresponds to the one previously presented and analyzed in Section 4.2.2, where the synchronization issue is detected during simulation.

```
SystemC MDVP 1.0.0
Copyright (C) 2012-2015 by all Contributors,
ALL RIGHTS RESERVED

----- Module B / Executing set_attributes()
----- Module A / Executing set_attributes()                    (1)

SC MDVP error                                                  (2)
Error: SystemC MDVP: Error elaborating the DE-TDF solver instantiated for the TDF
cluster containing the modules - B - A - : a valid TDF schedule cannot be completely
determined for this cluster because some synchronization problems are present.

TDF cluster information:                                       (3)
 |-- Cluster timestep = 12 ms
 |-- Modules:
 |   |-- name = B,  -- time step = 4 ms, -- calls per period = 3
 |   |-- ports:
 |   |   |-- name = B.in
 |   |   |-- time step = 2 ms
 |   |   |-- rate = 2
 |   |   |-- delay = 0
 |   |
 |   |   |-- name = B.out
 |   |   |-- time step = 4 ms
 |   |   |-- rate = 1
 |   |   |-- delay = 0
 |   |
 |   |-- name = A, -- time step = 6 ms, -- calls per period = 2
 |   |-- ports:
 |   |   |-- name = A.in
 |   |   |-- time step = 6 ms
 |   |   |-- rate = 1
 |   |   |-- delay = 0
 |   |
 |   |   |-- name = A.out
 |   |   |-- time step = 2 ms
 |   |   |-- rate = 3
 |   |   |-- delay = 0

Incomplete schedule determined for the TDF cluster:           (4)
 · t = 0 s · Read sig1
 · t = 0 s · A
 · t = 0 s · B
 · t = 0 s · Write sig3

Delay changes suggested for solving the synchronization problems found in TDF
converter ports during the elaboration phase:
    |-- port name         = B.out                              (5)
    |-- current delay     = 0
    |-- suggested delay = 1
```

Figure 6.19: Execution of the TDF Model shown in Figure 6.10, using SystemC MDVP.

On the other hand, in SystemC MDVP, only the elaboration phase is executed before detecting the synchronization issue:

(1) The execution of the set_attributes() function implemented inside each TDF module is performed.

(2) The synchronization issue is detected and notified to the designer. The detection is performed by means of the equivalent CPN constructed and analyzed during the TDF computability check stage.

The notification provided to the designer indicates that a valid TDF schedule cannot be found, shows the TDF cluster information (3) (attributes associated to each TDF component), shows the incomplete schedule determined before finding the synchronization issues (4), and indicates the delay changes proposed for solving the synchronization issues present in the TDF cluster (5).

The main advantage of the SystemC MDVP simulation is that the existing synchronization issues of a TDF cluster can be detected before starting simulation, it is before calling the initialize() and processing() functions implemented inside each TDF module. Besides, a single notification is provided to the designer, in order to summarize the delay changes suggested for solving all the synchronization issues of a TDF cluster. An example of a TDF cluster with several synchronization issues is discussed in Chapter 7.

6.6. Conclusion and Outlook

In this chapter, we introduced the implementation of a MoC according to the methodology, presented in Chapter 5, to add MoCs in SystemC MDVP. Initially, we defined the TDF MoC interface to be respected by TDF modules, and solvers which want to interact with TDF; and we specified how the most important functions of the TDF modules, TDF channels and TDF ports are implemented.

As the TDF MoC was the first MoC added to SystemC MDVP, we located it under the DE MoC in the SystemC MDVP architectural model. Therefore, we defined the TDF converter ports ensuring the data synchronization between the DE and TDF MoCs, and the DE-TDF solver ensuring the time synchronization between the same MoCs. This solver included the synchronization principles proposed in Chapter 4.

Using the SystemC MDVP TDF MoC, we implemented and simulated a basic TDF cluster, which includes interactions with the DE MoC. Once simulated, we compared the results provided by the MDVP simulator with the ones provided by SystemC-AMS. Thanks to the comparison, we demonstrated the main advantage of our DE-TDF synchronization approach, which is the detection of synchronization issues before simulation.

The proposed TDF MoC provides solid foundations, which can be extended to include all the functions defined in the SystemC AMS standard. For example, functions to embed linear dynamic equations in TDF modules, or functions to handle TDF modules' and ports' attributes changes during simulation.

Adding the TDF MoC in the SystemC MDVP simulator prototype, we have validated the DE-TDF synchronization approach introduced in Chapter 4, and the methodology to add MoCs interacting with the DE MoC. Additional models of computation should still be included in the simulator prototype to validate the genericity of the proposed approach.

CHAPTER

7 Case Study: Vibration Sensor

Contents

7.1. Introduction

In this chapter, in order to demonstrate the advantages of the Timed Data Flow (TDF) Model of Computation (MoC) implemented in the SystemC MDVP simulator prototype, and the advantages of the synchronization method proposed to ensure the interactions between the Discrete Event (DE) and Discrete Time (DT) domains, we present a case study of a vibration sensor model and its digital front end circuit, which includes a feedback loop and several interactions with the DE domain. This case study was inspired by the models presented in [69], [70].

In Section 7.2, we introduce the modeling of the case study. We describe the functionality of the model components, and the attributes associated to each one of them.

In Section 7.3, we detail how the model is elaborated. We present the validation of the rate attributes, the equivalent Coloured Petri Nets (CPN) model constructed, and the DE-TDF pre-simulation analysis applied to detect the DE-TDF synchronization issues.

In Section 7.4, we present and discuss the model elaboration and simulation results in two scenarios. First, when the model present several DE-TDF synchronization issues, we show that the simulator detects these issues and proposes the delay changes required. Second, when issues are not present, we show the execution trace of the model. In both scenarios, results are compared with the ones obtained with the SystemC-AMS proof-of-concept simulator.

Finally, in Section 7.5, we conclude this chapter.

7.2. Case Study Description

A vibration sensor and its digital front end circuit has been modeled using the TDF MoC. As shown in Figure 7.1, this model is composed of six TDF modules, some of them with multi-rate attributes; and one DE module, involved in a control closed loop. In addition, it handles interactions with the DE domain by means of the input and output converter ports instantiated in TDF modules.

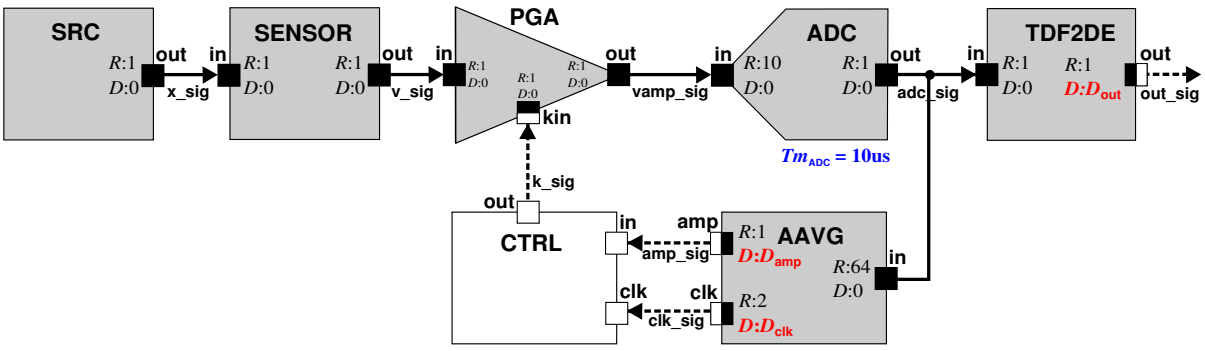

Figure 7.1: Vibration Sensor Model and its Digital Front End Circuit.

- The source **SRC** is modeled by means of a TDF module, which generates a vibration signal as a sequence of sinusoidal wavelets (representing a displacement **x_sig** in meters). The amplitude (4 µm) and the offset (−8 µm) of the generated signal are constant, and its oscillation frequency can take one of the three values: 2 kHz, 4 kHz, or 8 kHz.

- The vibration sensor **SENSOR** is modeled by means of a TDF module, which takes as input, the displacement (**x_sig**) caused by the vibration; and generates as output, a voltage signal (**v_sig**) proportional to the vibration velocity. This voltage signal is determined using the Equation 7.1.

$$\mathbf{v\_sig} = \mathbf{k_{trans}} \cdot \mathbf{x\_sig}' \qquad with \quad \mathbf{k_{trans}} = 1\,\mathrm{V\,s\,m^{-1}} \tag{7.1}$$

- The programmable gain amplifier **PGA** is modeled by means of a TDF module, which amplifies the input voltage signal (**v_sig**) by a gain ($\mathbf{2^{kin}}$). This gain is controlled by an input factor (**kin**) read from the DE domain. In this module, the output (**vamp_sig**) is saturated when the amplified voltage exceeds a supply voltage (**v_max** = 5 V). The amplified voltage is determined using the Equation 7.2. This equation indicates that $-5\,\mathrm{V} \leq \mathbf{vamp\_sig} \leq 5\,\mathrm{V}$.

$$\mathbf{vamp\_sig} = \begin{cases} \mathbf{-v\_max} & \mathbf{2^{kin} \cdot v\_sig} < \mathbf{-v\_max} \\ \mathbf{2^{kin} \cdot v\_sig} & \mathbf{-v\_max} \leq \mathbf{2^{kin} \cdot v\_sig} \leq \mathbf{v\_max} \\ \mathbf{v\_max} & \mathbf{2^{kin} \cdot v\_sig} > \mathbf{v\_max} \end{cases} \tag{7.2}$$

- The analog to digital converter **ADC** is modeled by means of a TDF module, which digitizes the amplified voltage (**vamp_sig**) in a n-bits integer (with n = 5), in which the most significant bit correspond to the sign bit. This integer is later transmitted by a TDF signal (**adc_sig**). It is the unique module in the system, in which the time step attribute $Tm_{\mathrm{ADC}} = 10\,\mu\mathrm{s}$ is assigned. In this module, the output (**adc_sig**) is determined using the Equation 7.3.

$$\mathbf{adc\_sig_{n-bits}} \approx \left(\frac{\mathbf{vamp\_sig}}{\mathbf{v\_max}} \right) \cdot \mathbf{2^{n-1}} \tag{7.3}$$

- The TDF to DE converter **TDF2DE** is modeled by means of a TDF module, which forwards the digitized value ($\mathbf{adc\_sig_{n-bits}}$) to the DE domain, using a DE signal ($\mathbf{out\_sig_{n-bits}}$).

- The amplitude estimator **AAVG** is modeled by means of a TDF module, which calculates the absolute average amplitude of the $\boldsymbol{n_s}$ received samples ($\boldsymbol{n_s}$ = 64, which is the rate attribute associated to the input port of the **AAVG** module). The absolute average amplitude (**amp_sig**) is determined using the Equation 7.4.

$$\mathbf{amp\_sig_{n-bits}} = \frac{\mathbf{1}}{\boldsymbol{n_s}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n_s}} \mathbf{adc\_sig}_{\boldsymbol{i}} \tag{7.4}$$

- The gain controller **CTRL** is modeled by means of a DE module, which implements the Finite-State Machine (FSM) shown in Figure 7.2. This FSM varies a factor (k) according to the estimated amplitude (**amp_sig**), and the threshold values determined using the Equations 7.5 and 7.6. This factor (k) is transmitted to the **PGA** module, using a DE signal (**k_sig**).

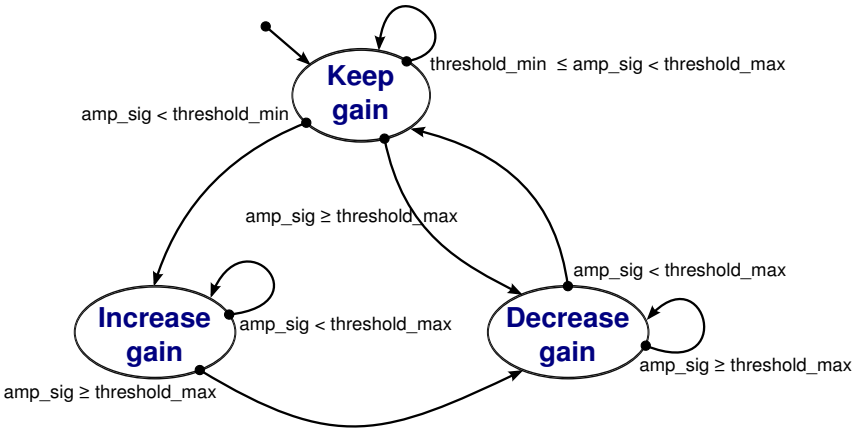

Figure 7.2: Finite-State Machine of Gain Controller.

$$\textbf{threshold\_min} = \mathbf{20\%} \cdot \mathbf{2^{n-1}} \tag{7.5}$$

$$\textbf{threshold\_max} = \mathbf{60\%} \cdot \mathbf{2^{n-1}} \tag{7.6}$$

The particularity of this model is that its multi-rate TDF cluster becomes part of a closed loop including a path through the DE domain. As the TDF cluster itself contains no loops, it could be assumed that port delay assignments are not necessary to calculate a valid schedule [28], but this is only valid for single-rate TDF models. In order to demonstrate this condition, and evaluate the detection of synchronization issues in this model, delay attributes are not assigned in TDF ports ($D_{\text{out}} = 0$, $D_{\text{amp}} = 0$ and $D_{\text{clk}} = 0$).

More details about the implementation of the vibration sensor model shown in Figure 7.1 are presented in Appendix A.

7.3. Model Elaboration

7.3.1. Creation of Clusters and Instantiation of Solvers

During the first stage of the SystemC MDVP elaboration, one TDF cluster is identified in the model shown in Figure 7.1. It contains six TDF modules (**SRC**, **SENSOR**, **PGA**, **ADC**, **TDF2DE** and **AAVG**); and it will be executed following the time synchronization constraints imposed by its master MoC (DE).

As the pair of master-slave MoCs identified in the model is the <DE-TDF> pair, the simulator instantiates a **DE-TDF solver** on the identified cluster. This solver, as introduced in Chapter 6, is the responsible of executing the TDF elaboration and simulation phases. The representation of the hierarchy of solvers created by the simulator for the model previously presented, is shown in Figure 7.3.

7.3.2. Elaboration of Modules by means of Solvers

When the SystemC MDVP kernel calls the elaborate() method on the **TDF-DE solver**, the TDF elaboration phase runs, according to the different phases presented in Section 6.3.1. First, the attributes specified

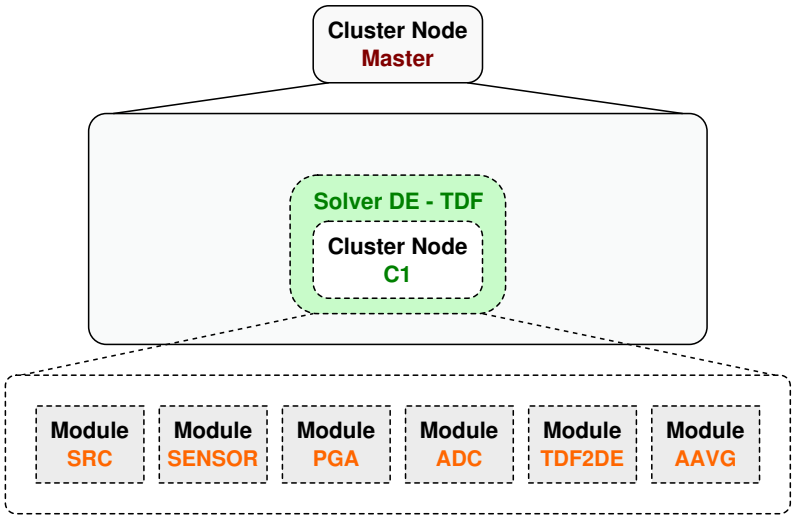

Figure 7.3: Hierarchy of Solvers Constructed for the Model shown in Figure 7.1.

by the designer are assigned to the TDF modules and ports. For example, the time step $Tm_{\text{ADC}} = 10\,\mu s$, specified in the **ADC** module; the rate attribute $R = 64$, specified in the input TDF port **in** of module **AAVG**; or the rate attribute $R = 2$, specified in the output converter port **clk** of the same module.

Second, once the attributes have been assigned, the simulator verifies that at least one TDF module has a time step attribute assigned, in this case, the **ADC** module. Therefore, this time step is propagated to the remaining TDF modules and ports instantiated in the same identified cluster. The results of propagation are illustrated in Figure 7.4.

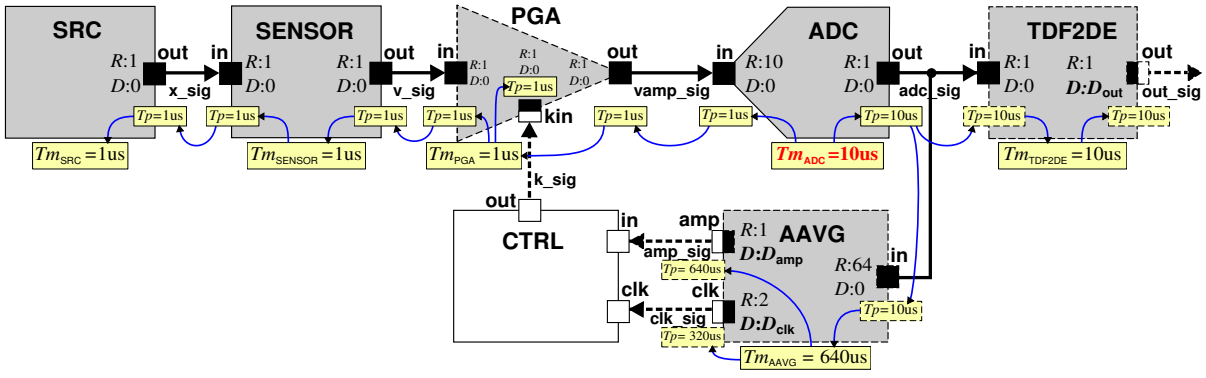

Figure 7.4: Time Step Propagation inside the Model shown in Figure 7.1.

Third, the TDF cluster computability is checked. By means of the analysis based on the Synchronous Data Flow (SDF) formalism, the rate consistencies are verified; and the number of times (q) that each TDF module should be executed in the cluster period ($Tcls$) are calculated. In addition, by means of the analysis based on Coloured Petri Nets (CPN), the synchronization issues, which arise when the TDF cluster interacts with the DE domain, are detected. A solution is proposed to the designer. The TDF cluster computability check is described below.

a. Analysis based on the SDF formalism

Isolating the TDF cluster from the DE domain, it can be represented using the SDF graph, shown in Figure 7.5. Based on this graph, the topology matrix $\Gamma_{i,j}$ is constructed (Equation 7.7), and its rank is calculated (Equation 7.8). In the matrix, each (i, j)th entry is the amount of data produced by a node j (**SRC**, **SENSOR**, **PGA**, **ADC**, **TDF2DE**, **AAVG**) on an arc i (**x_sig**, **v_sig**, **vamp_sig**, **adc_sig**$_{\text{TDF2DE}}$, **adc_sig**$_{\text{AAVG}}$).

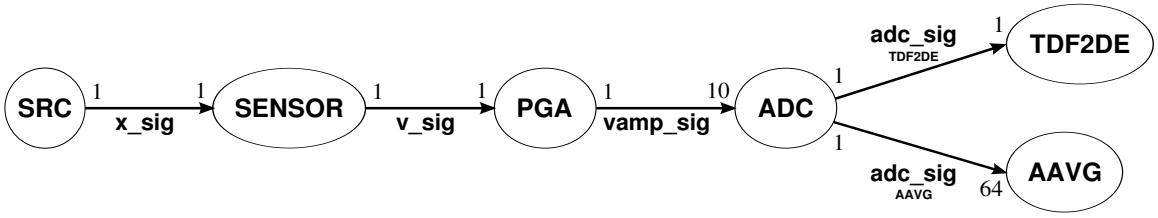

Figure 7.5: SDF Graph of the Isolated TDF Cluster Identified for the Model shown in Figure 7.1.

$$\Gamma_{i,j} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -64 \end{bmatrix} \tag{7.7}$$

As the rank of the $\Gamma_{i,j}$ matrix is equal to $N-1$ ($N = 6$ is the number of nodes is the SDF graph), the rate consistencies are confirmed in the model.

$$Rank\left(\Gamma_{i,j}\right) = 5 = N - 1 \tag{7.8}$$

Therefore the number of executions $q_{j,1}$ of each TDF module in a cluster period, are determined as shown in Equation 7.9.

$$\Gamma_{i,j} \cdot q_{j,1} = 0 \tag{7.9}$$

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -64 \end{bmatrix} \cdot \begin{bmatrix} q_{\text{SRC}} \\ q_{\text{SENSOR}} \\ q_{\text{PGA}} \\ q_{\text{ADC}} \\ q_{\text{TDF2DE}} \\ q_{\text{AAVG}} \end{bmatrix} = 0$$

$$\begin{bmatrix} q_{\text{SRC}} \\ q_{\text{SENSOR}} \\ q_{\text{PGA}} \\ q_{\text{ADC}} \\ q_{\text{TDF2DE}} \\ q_{\text{AAVG}} \end{bmatrix} = \begin{bmatrix} 640 \\ 640 \\ 640 \\ 64 \\ 64 \\ 1 \end{bmatrix}$$

Finally, the TDF cluster period is calculated as shown in Equation 7.10.

$$Tcls = Tm_{\mathrm{j}} \cdot q_{\mathrm{j}} \tag{7.10}$$

$$Tcls = Tm_{\mathrm{SRC}} \cdot q_{\mathrm{SRC}} = Tm_{\mathrm{ADC}} \cdot q_{\mathrm{ADC}} = Tm_{\mathrm{AAVG}} \cdot q_{\mathrm{AAVG}}$$
$$Tcls = 1\,\mu s \cdot 640 = 10\,\mu s \cdot 64 = 640\,\mu s \cdot 1$$
$$Tcls = 640\,\mu s$$

Note: In the simulator, this phase is performed using the Eigen library [71].

b. Analysis based on CPN

Considering the transformation rules presented in Section 4.3.2, the TDF cluster is represented by means of a timed CPN, shown in Figure 7.6. In this model:

- The number of read synchronization operations to be performed by the TDF input converter port **kin** of module **PGA** is $read_{ops_{\mathrm{kin}}} = q_{\mathrm{PGA}} * R_{\mathrm{kin}} = 640 * 1 = 640$.
- The number of write synchronization operations to be performed by the TDF output converter port **out** of module **TDF2DE** is $write_{ops_{\mathrm{out}}} = q_{\mathrm{TDF2DE}} * R_{\mathrm{out}} = 64 * 1 = 64$.
- The number of write synchronization operations to be performed by the TDF output converter port **amp** of module **AAVG** is $write_{ops_{\mathrm{amp}}} = q_{\mathrm{AAVG}} * R_{\mathrm{amp}} = 1 * 1 = 1$.
- The number of write synchronization operations to be performed by the TDF output converter port **clk** of module **AAVG** is $write_{ops_{\mathrm{amp}}} = q_{\mathrm{AAVG}} * R_{\mathrm{amp}} = 1 * 2 = 2$.

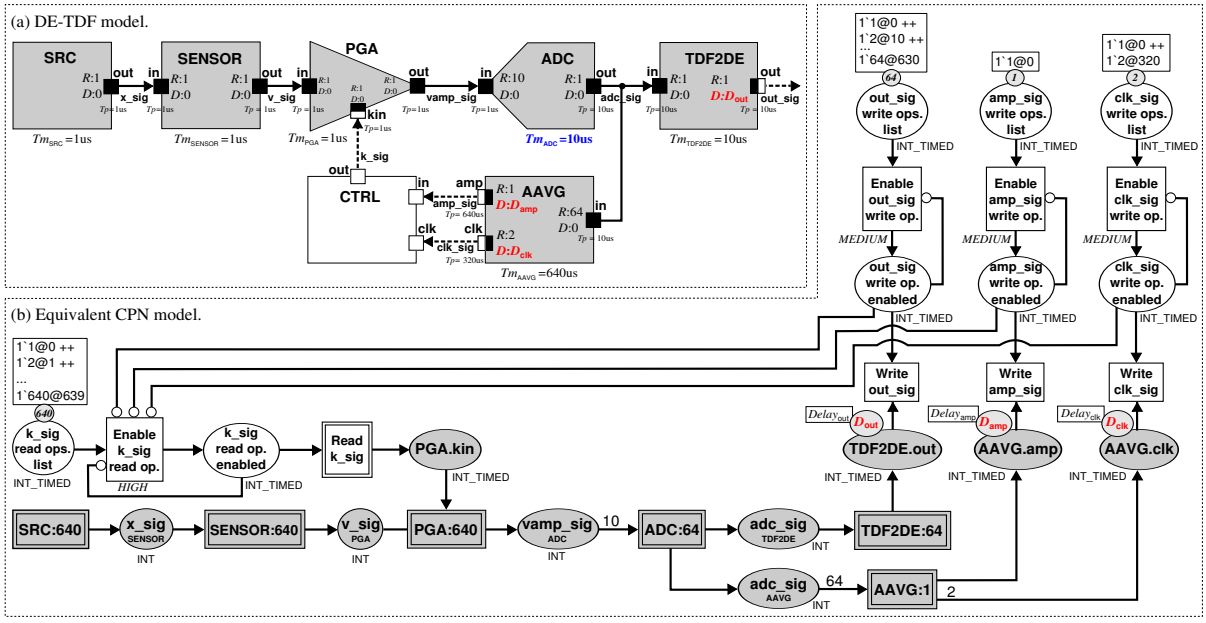

Figure 7.6: Vibration Sensor Model and its Equivalent CPN Model.

When the simulator executes the analysis, presented in Section 4.4, on the previous timed CPN equivalent model with delay attributes $D_{\mathrm{out}} = 0$, $D_{\mathrm{amp}} = 0$ and $D_{\mathrm{clk}} = 0$, the causality problems are detected during elaboration.

First, at $t_{CPN} = 0\mu s$, when the transitions **Read k_sig**, **SRC**, **SENSOR**, **SRC**, **PGA**, **SENSOR**, and **SRC** have been executed and added to the schedule, the CPN is locked and has not reached its final state (it is shown with blue inscriptions in Figure 7.7). Then, the simulator detects several synchronization problems, as shown in the yellow block of Figure 7.7:

- The places **out_sig write op. enabled**, **amp_sig write op. enabled** and **clk_sig write op. enabled** indicate that three write synchronization operations should be performed at $t_{CPN} = 0\mu s$.
- The places **TDF2DE.out**, **AAVG.amp** and **AAVG.clk** have no tokens to be consumed at time $t_{CPN} = 0\mu s$.
- Therefore, the transitions **Write out_sig**, **Write amp_sig** and **Write clk_sig** are locked, then the write synchronization operations cannot be performed.

In order to continue with the analysis for detecting all the synchronization problems in the model, the simulator temporarily fixes these problems, following the approach presented in Section 4.4.4:

- The tokens "1@0" are deleted from the **out_sig write op. enabled**, **amp_sig write op. enabled** and **clk_sig write op. enabled** places.
- The delay attribute in the **TDF2DE.out**, **AAVG.amp**, and **AAVG.clk** places is increased to $D_{\mathsf{out}} = 1$, $D_{\mathsf{amp}} = 1$ and $D_{\mathsf{clk}} = 1$.

After delay modifications, the model analysis continues until the CPN is locked again, without finding its final state (it is shown with blue inscriptions in Figure 7.8). Then, the simulator detects a new synchronization problem at $t_{CPN} = 320\mu s$, as shown in the yellow block of Figure 7.8:

- The place **clk_sig write op. enabled** indicates that one write synchronization operation should be performed at $t_{CPN} = 320\mu s$.
- The place **AAVG.clk** has no tokens to be consumed at time $t_{CPN} = 320\mu s$.
- Therefore, the transition **Write clk_sig** is locked, then the write synchronization operation cannot be performed.

The detected problem is temporarily solved deleting the token "2@320" from the **clk_sig write op. enabled** place, and increasing the delay attribute in the **AAVG.clk** place to $D_{\mathsf{clk}} = 2$. After this modification, the model analysis is completed for the TDF cluster period $Tcls = 640\mu s$.

At the end of the analysis, as synchronization problems were detected, the model is not considered computable, the schedule cannot be defined, and the delay attributes changes are notified to the designer, as shown in Figure 7.9. This determined delay changes, $D_{\mathsf{out}} = 1$, $D_{\mathsf{amp}} = 1$ and $D_{\mathsf{clk}} = 2$, are required to solve the causality problems in the model. Using this information, the designer can modify the model and restart its execution.

If, in comparison to our approach, the unmodified TDF model is run in the SystemC-AMS proof-of-concept simulator, the errors will be detected one by one during simulation. Thus, the designer needs to perform three complete simulations to determine all required delay attribute changes.

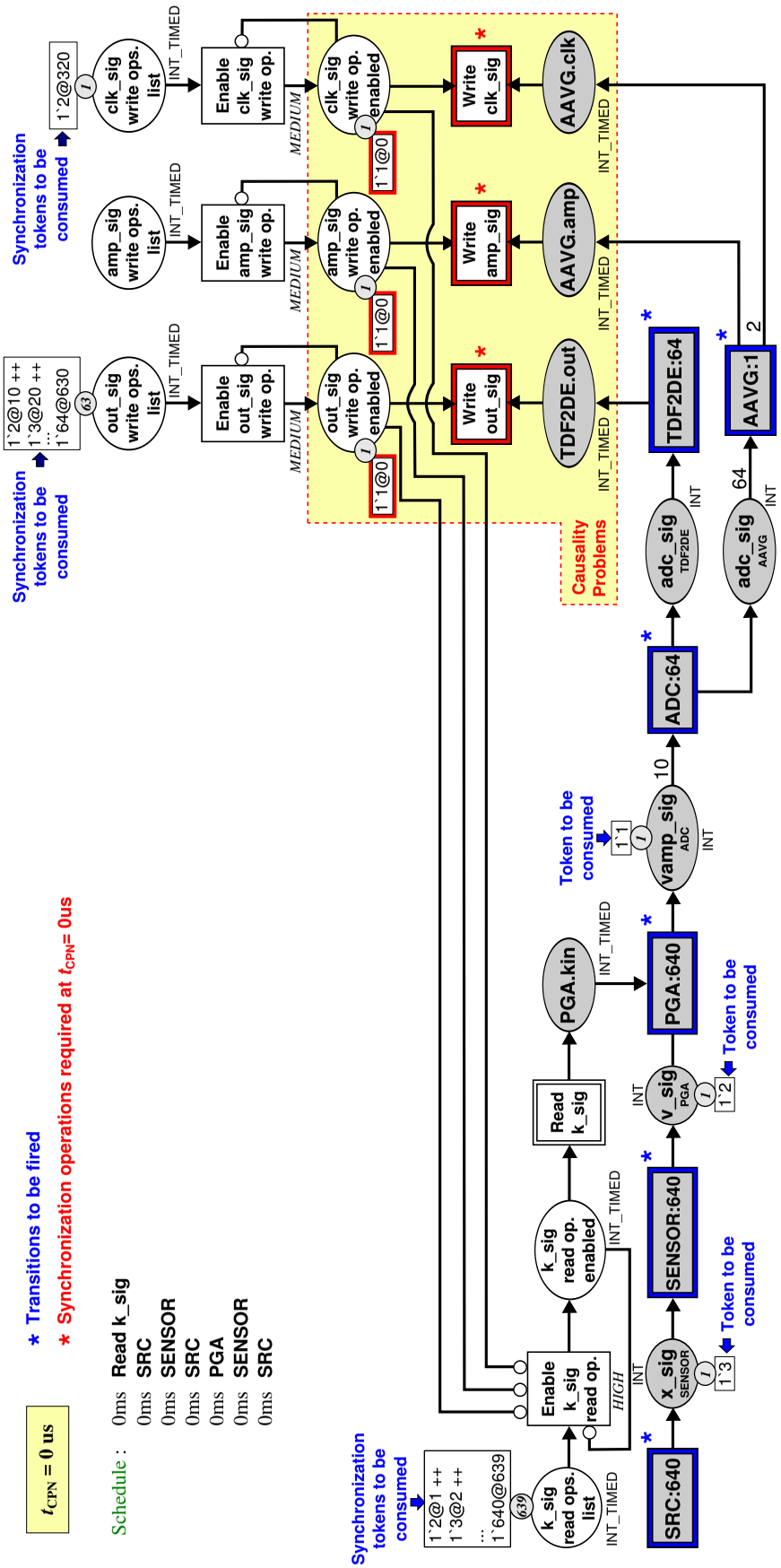

Figure 7.7: First Detection of Synchronization Problems, at $t_{CPN} = 0\,\mu s$, in the Equivalent CPN Model shown in Figure 7.6, with $D_{out} = 0$, $D_{amp} = 0$ and $D_{clk} = 0$.

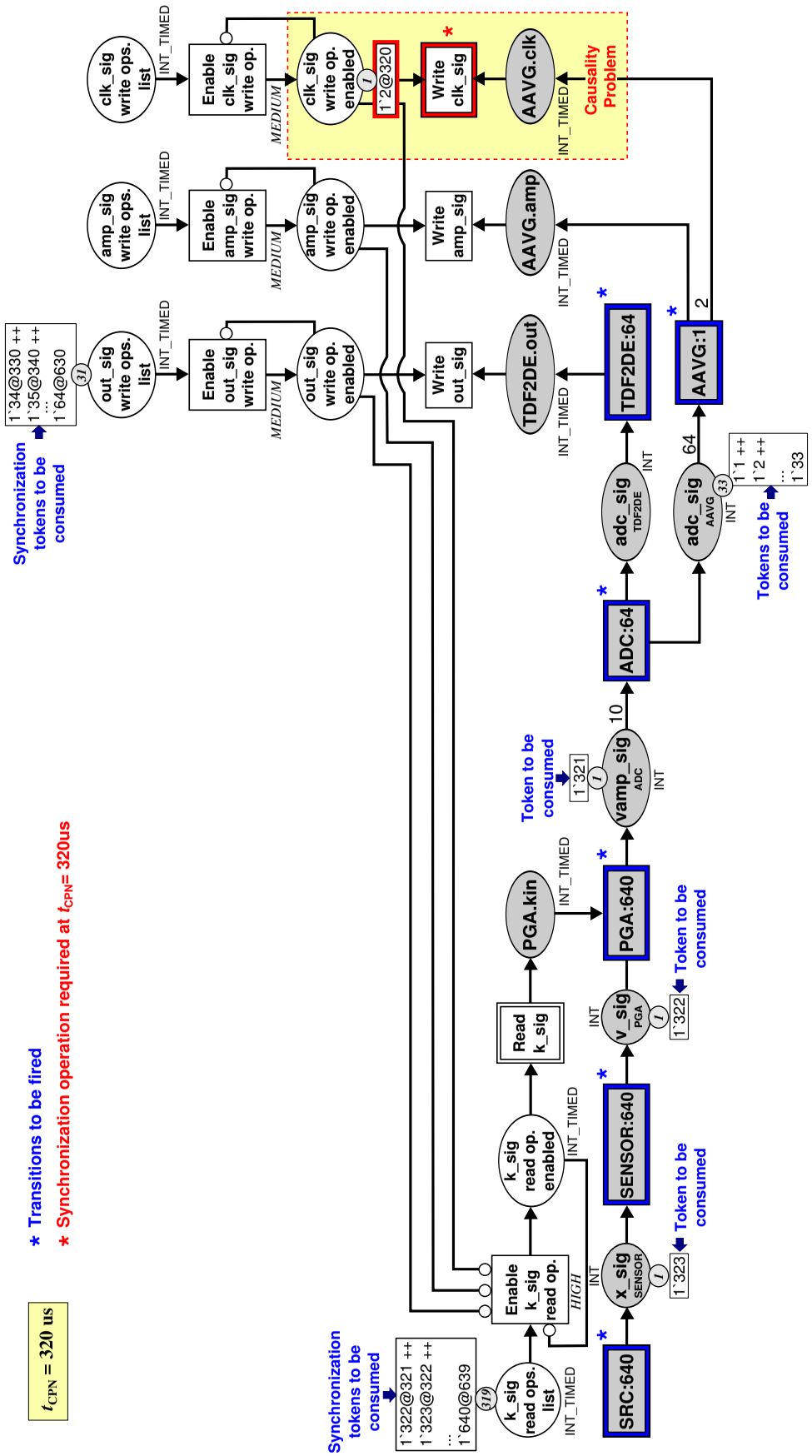

Figure 7.8: Second Detection of Synchronization Problems, at $t_{\text{CPN}} = 320\,\mu\text{s}$, in the Equivalent CPN Model shown in Figure 7.6, with $D_{\text{out}} = 0$, $D_{\text{amp}} = 0$ and $D_{\text{clk}} = 0$.

```
SystemC MDVP 1.0.0
Copyright (C) 2012-2015 by all Contributors,
ALL RIGHTS RESERVED
SC MDVP error
Error: SystemC MDVP: Error elaborating the TDF-DE solver instantiated for the TDF
cluster containing the modules - TDF2DE - AAVG - ADC - PGA - SENSOR - SRC - :
a valid TDF schedule cannot be completely determined for this cluster because
some synchronization problems are present.

TDF cluster information:
 |-- Cluster timestep = 640 us
 |-- Modules:
...
 |   |-- name = AAVG
 |   |-- time step = 640 us
 |   |-- calls per period = 1
 |   |-- ports:
 |   |   |-- name = AAVG.in
 |   |   |-- time step = 10 us
 |   |   |-- rate = 64
 |   |   |-- delay = 0
 |   |   |
 |   |   |-- name = AAVG.clk
 |   |   |-- time step = 320 us
 |   |   |-- rate = 2
 |   |   |-- delay = 0
 |   |   |
 |   |   |-- name = AAVG.out
 |   |   |-- time step = 640 us
 |   |   |-- rate = 1
 |   |   |-- delay = 0
...

Incomplete schedule determined for the TDF cluster:
 · t = 0 s · Read k_sig
 · t = 0 s · SRC
 · t = 0 s · SENSOR
 · t = 0 s · SRC
 · t = 0 s · PGA
 · t = 0 s · SENSOR
 · t = 0 s · SRC

Delay changes suggested for solving the synchronization problems found in TDF
converter ports during the elaboration phase:
    |-- port name          = TDF2DE.out
    |-- current delay      = 0
    |-- suggested delay = 1

    |-- port name          = AAVG.clk
    |-- current delay      = 0
    |-- suggested delay = 2

    |-- port name          = AAVG.out
    |-- current delay      = 0
    |-- suggested delay = 1
```

Figure 7.9: Execution of the Vibration Sensor Model (with $D_{\mathsf{out}} = 0$, $D_{\mathsf{amp}} = 0$ and $D_{\mathsf{clk}} = 0$) Using SystemC MDVP.

7.4. Model Simulation

Once the model has been modified by the designer (to set $D_{\mathsf{out}} = 1$, $D_{\mathsf{amp}} = 1$ and $D_{\mathsf{clk}} = 2$), and the execution is restarted, the SystemC MDVP simulator:

- Determines during elaboration the schedule, which includes the order at which the TDF modules and its DE interactions should be executed.
- Initializes the TDF modules with delay attributes associated, as introduced in Section 6.3.2.a.
- Performs the registration of the determined schedule in the SystemC DE kernel, as introduced in Section 6.3.2.b.

Figure 7.10 shows the results obtained once the schedule previously registered (by SystemC MDVP) is executed under the control of the SystemC DE kernel. These results match the ones obtained when the model is run in the SystemC-AMS proof-of-concept simulator.

- The source generates a sinusoidal signal ($\mathsf{x}_{\mathsf{sig}}$) with amplitude 4 µm, offset −8 µm and frequencies between 2 kHz, 4 kHz, and 8 kHz, which represents the vibration displacement.
- The vibration sensor generates a voltage signal ($\mathsf{v}_{\mathsf{sig}}$) proportional to the vibration velocity.
- This voltage signal is amplified ($\mathsf{vamp}_{\mathsf{sig}}$) by a factor of gain ($2^{\mathsf{ksig}}$), and digitized ($\mathsf{adc}_{\mathsf{sig}}$). The amplified and digitized voltage are saturated when they exceed ±5 V.
- The DE signal ($\mathsf{amp}_{\mathsf{sig}}$) represents the absolute average amplitude for every 64 samples received from the **ADC**.

7.5. Conclusion

In this chapter, we presented a case study of a vibration sensor model and its digital front end circuit, which includes a feedback loop and several interactions with DE domain. We demonstrated that:

- The synchronization issues arising in TDF clusters interacting with the DE domain can be detected and resolved before simulation.
- A single notification is provided to the designer, in order to summarize the delay changes suggested for solving all the synchronization issues of a TDF cluster.
- Multi-rate clusters, which include DE loops, require port delay assignments to calculate a valid schedule.

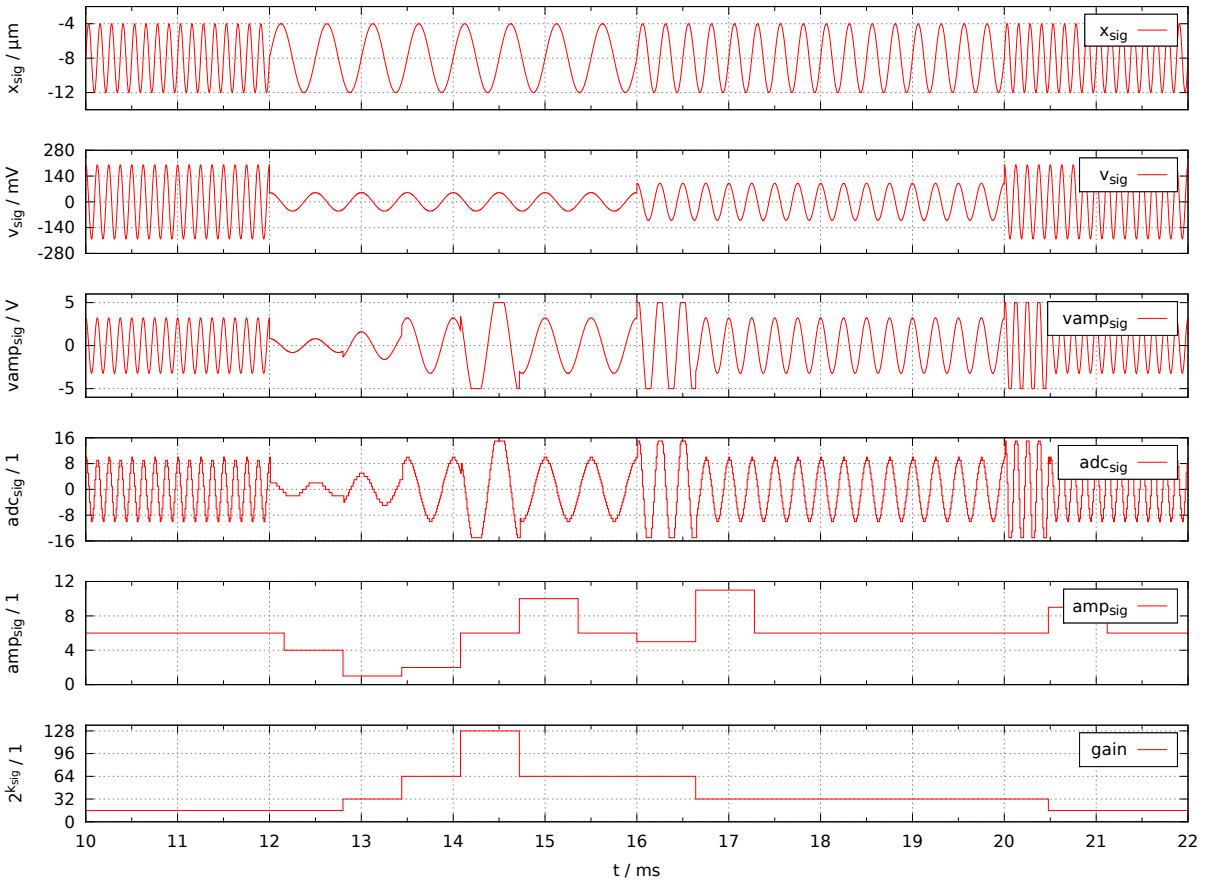

Figure 7.10: SystemC MDVP Simulation Traces of the Vibration Sensor Model with $D_{\mathsf{out}} = 1$, $D_{\mathsf{amp}} = 1$ and $D_{\mathsf{clk}} = 2$.

CHAPTER

8 Conclusion

Contents

8.1. Conclusion

In this thesis, we explored the possibilities of modeling, simulating, and synchronizing multi-disciplinary systems with respect to the Discrete Event (DE) domain, using as reference the SystemC Analog/Mixed Signal (AMS) simulation standard [13]. We analyzed the SystemC-AMS proof-of-concept simulator and identified the issues limiting its extension:

- The addition of Models of Computation (MoCs) is in the hands of experts.
- The interactions with DE are handled through only one synchronization method. This method is defined between the SystemC DE simulation kernel and the Timed Data Flow (TDF) MoC.

Regarding this unique synchronization method, we identified some drawbacks:

- The detection of synchronization errors is performed during the simulation phase.
- We miss a formalized method to analyze the interactions and the occurrence of the synchronization errors between DE and TDF.

In order to provide a solution to the identified issues, we introduced and implemented a new simulator prototype called SystemC Multi-Disciplinary Virtual Prototyping (MDVP). In this thesis, the issues were addressed as presented below.

- **Analysis and formalization of DE-TDF interactions**: in Chapter 4, we identified the different timescales handled during the execution of TDF models; described how the synchronization problems can arise between DE and TDF; proposed a method for representing DE-TDF clusters using Coloured Petri Nets (CPN); and introduced a DE-TDF pre-simulation analysis method for determining, in advance, when the TDF clusters interact with the DE domain during simulation.

  Thanks to this analysis, when DE-TDF models do not have synchronization problems, SystemC MDVP can determine and register in the SystemC DE simulation kernel, before simulation, the execution order between the TDF modules and their interactions with the DE domain. It ensures that DE-TDF models can be executed without interruptions once the simulation begins.

- **Detection of DE-TDF synchronization problems**: using the DE-TDF pre-simulation analysis, SystemC MDVP can also identify the synchronization problems, which can arise in TDF models interacting with the DE domain, and determine the required delay attribute changes to solve the timing inconsistencies.

  The advantage in SystemC MDVP is that designers are notified of all the existing problems before starting simulation.

- **Synchronization, elaboration and simulation of MoCs**: in Chapter 5, we introduced the SystemC MDVP hierarchical synchronization approach, which is based on the principle that two different MoCs can be synchronized if, and only if, at least one synchronization method is defined to handle the different timescales involved between them. This means that several synchronization methods can be defined to perform the direct interaction of MoCs with the

DE domain, and to perform the interactions between different pairs of MoCs, without being limited by the discrete time semantics implemented by the TDF MoC.

Besides, we defined new generic elaboration and simulation phases, which identify the clusters of a model, detect the pair of master-slave MoCs associated to each cluster, and instantiate the solver responsible for handling the elaboration, simulation and synchronization of the cluster's components.

The advantage is that the identification and instantiation of solvers in multi-disciplinary models is automatically performed by the simulator.

- **Addition of MoCs**: in Chapter 5, we also defined a methodology to add models of computation in SystemC MDVP. This methodology includes the definition of the abstract methods, which allow the MoC elaboration and simulation; the specification of the MoC components (modules, ports and channels) that will be provided to the designer; and the selection of the models of computation with which the MoC being defined wants to interact. Once these MoCs are selected, additional MoC components should be defined: converter ports, to handle the data synchronization; and solvers to handle the time synchronization.

  The advantage in SystemC MDVP is that the addition of MoCs does not modify the *generic elaboration and simulation phases* defined by this simulator.

The proposed solutions were validated in Chapter 6, by means of the addition of a TDF MoC in SystemC MDVP. This MoC was designed to directly communicate with the DE domain, it implemented the DE-TDF pre-simulation analysis introduced in Chapter 4, and it was added to SystemC MDVP following the methodology proposed in Chapter 5.

In addition, in Chapter 7, a case study was presented to demonstrate the advantages of the TDF MoC implemented:

- The synchronization issues arising in TDF clusters interacting with the DE domain can be detected and resolved before simulation.
- A single notification is provided to the designer, in order to summarize the delay changes suggested for solving all the synchronization issues of a TDF cluster.
- Multi-rate clusters, which include DE loops, require port delay assignments to calculate a valid schedule.

8.2. Future Work

The presented work introduced the principles used to define and implement the SystemC MDVP simulator prototype. Based on these principles, the following topics can be investigated:

- **Extension of the TDF MoC**: in order to provide all the TDF features defined in the SystemC AMS standard, the TDF MoC should be extended to include:

  – The ability to model Continuous Time (CT) behaviors inside a TDF module, by means of the definition of linear transfer functions on the Laplace domain, or state-space equations. These functions or equations should be handled by a TDF-CT solver able to provide solutions at the discrete time synchronization points imposed according to the TDF sematics.

  - Dynamic TDF features to allow the modification of the TDF module attribute (timestep) and the TDF port attributes (timestep, rate and delay) during simulation. To this end, new abstract functions should be defined and included in the TDF elaboration and simulation phases proposed for SystemC MDVP. Besides, a method should be implemented to re-execute the attribute settings, time step calculation and propagation, and computability check stages, after performing changes in a model during simulation. The cost of re-executing these stages should be carefully evaluated.

- **Implementation of new MoCs**: in order to validate the genericity of the approach presented in this thesis, new MoCs, described in different time domains, should be defined and added to SystemC MDVP. At present, the addition of two MoCs in SystemC MDVP is being investigated in the framework of the European project CATRENE CA701 Heterogeneous Inception (H-INCEPTION) [66]. First, the BG (Bond Graph) MoC, designed for the description of conservative (energy conserving) behavior. Second, the SPH (Smoothed-Particle Hydrodynamics) MoC designed to describe and simulate fluid flows.

- **Implementation of new features in SystemC MDVP**: alike the computability checks of dimensions and units included in a model, the functional verification of properties in models, and the implementation of monitoring and tracing mechanisms. These aspects are being investigated in the framework of another thesis work [67].

Bibliography

[1] acatech – National Academy of Science and Engineering, Ed., *Cyber-Physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*. Munich, Germany: acatech, 2011. [Online]. Available: http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Publikationen/Stellungnahmen/acatech_POSITION_CPS_Englisch_WEB.pdf.

[2] L. Coetzee and J. Eksteen, "The Internet of Things - Promise for the Future? An Introduction", in *Proceedings of the IST-Africa Conference*, Gaborone, Botswana: IEEE, 2011, pp. 1–9, ISBN: 978-1-4577-1077-3.

[3] *Gartner Says 4.9 Billion Connected "Things" Will Be in Use in 2015*, Barcelona, Spain, 2014. [Online]. Available: http://www.gartner.com/newsroom/id/2905717.

[4] *Cyber-Physical Systems (CPS). Program Solicitation NSF 15-541*, Virginia, USA, 2015. [Online]. Available: http://www.nsf.gov/pubs/2015/nsf15541/nsf15541.htm.

[5] B. Brech, J. Jamison, L. Shao, and G. Wightwick, *The Interconnecting of Everything. An IBM Redbooks®Point-of-View publication by the IBM Academy of Technology*, USA, 2013. [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp4975.html?Open.

[6] "Cyber-Physical Systems – Merging the physical and virtual worlds", in *Cyber-Physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*, acatech – National Academy of Science and Engineering, Ed., Munich, Germany: acatech, 2011, pp. 15–22. [Online]. Available: http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Publikationen/Stellungnahmen/acatech_POSITION_CPS_Englisch_WEB.pdf.

[7] "Virtual Prototypes: The Engine Behind "Shift left"", in *Better Software. Faster!: Best Practices in Virtual Prototyping*, De Schutter, Tom, Ed., Mountain View, CA, USA: Synopsys, Inc., 2014, pp. 17–41.

[8] J. Moreno, M. Damm, J. Haase, C. Grimm, and E. Holleis, "Unified and Comprehensive Electronic System Level, Network and Physics Simulation for Wirelessly Networked Cyber Physical Systems", in *Forum on specification and Design Languages (FDL)*, Vienna, Austria: ECSI - European Electronic Chips and Systems Design Initiative, 2012, ISBN: 978-2-9530504-6-2. [Online]. Avail-

able: http://www.ecsi.org/resource/fdl/2012/paper/unified-and-comprehensive-electronic-system-level-network-and-physics-simulation-wirelessly-networke.

[9] F. Bouchhima, M. Brière, G. Nicolescu, M. Abid, and E. Aboulhamid, "A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation", in *Proceedings of the IEEE International Behavioral Modeling and Simulation Workshop*, San José, CA: IEEE, 2006, pp. 1–6, ISBN: 0-7803-9742-8. DOI: 10.1109/BMAS.2006.283461.

[10] M. Barnasconi, *SystemC AMS Extensions: Solving the Need for Speed*, San Jose, CA, USA, 2010. [Online]. Available: http://accellera.org/resources/articles/amsspeed.

[11] C. Grimm, M. Damm and J. Haase, "Towards Co-design of HW/SW/Analog Systems", in *Design Methodologies for Secure Embedded Systems*, Alexander Biedermann and H. Gregor Molter, Ed., Darmstadt, Germany: Springer Berlin Heidelberg, 2011, pp. 1–24, ISBN: 978-3-642-16766-9. DOI: 10.1007/978-3-642-16767-6.

[12] N. Bajaj, P. Nuzzo, M. Masin, and A. Sangiovanni-Vincentelli, "Optimized selection of reliable and cost-effective cyber-physical system architectures", in *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, Grenoble, France, 2015, pp. 561–566, ISBN: 978-3-9815370-4-8.

[13] Accellera SystemC AMSWG, *Standard SystemC AMS Extensions Language Reference Manual*, version 2.0, Accellera Systems Initiative, 2013. [Online]. Available: http://www.accellera.org/.

[14] IEEE Computer Society, *1666-2011 IEEE Standard for SystemC Language Reference Manual*, IEEE, 2012, ISBN: 978-0-7381-6801-2 STD97162.

[15] A. Jantsch and I. Sander, "Models of computation and languages for embedded system design", *IEE Proceedings on Computers and Digital Techniques*, vol. 152, no. 2, pp. 114–129, 2005. DOI: 10.1049/ip-cdt:20045098.

[16] Fraunhofer IIS, *SystemC AMS 2.0 Proof-of-Concept Implementation*, Germany, 2014. [Online]. Available: http://www.systemc-ams.org/.

[17] K. Jensen and L. M. Kristensen, *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*. Springer-Verlag Berlin Heidelberg, 2009, ISBN: 978-3-642-00284-7. DOI: 10.1007/b95112.

[18] A. Leveque, F. Pêcheux, M.-M. Louërat, H. Aboushady, and M. Vasilevski, "SystemC-AMS Models for Low-Power Heterogeneous Designs: Application to a WSN for the Detection of Seismic Perturbations", in *23rd International Conference on Architecture of Computing Systems (ARCS))*, Hannover, Germany: VDE, 2010, pp. 1–6, ISBN: 978-3-8007-3222-7.

[19] M. Keating and P. Bricaud, "The System-On-Chip Design Process", in *Reuse Methodology Manual for System-on-a-Chip Designs*, USA: Kluwer Academic Publishers, 2002, pp. 9–22, ISBN: 978-0-306-47640-2. DOI: 10.1007/0-306-47640-1_2.

[20] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, 2nd edition. USA: Springer Science+Business Media, 2009, ISBN: 978-0-387-69957-8. DOI: 10.1007/978-0-387-69958-5.

[21] J. Bhasker, *A SystemC Primer*. USA: Star Galaxy Publishing, 2002, ISBN: 0-9650391-8-8.

[22] T. Grotker, S. Liao, M. Grant, and S. Swan, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002, ISBN: 1-4020-7072-1.

[23] IEEE Computer Society, "Introduction to TLM-2.0", in *1666-2011 IEEE Standard for SystemC Language Reference Manual*, New York, USA: IEEE, 2012, pp. 415–425, ISBN: 978-0-7381-6801-2 STD97162.

[24] D. C. Black, J. Donovan, B. Bunton, and A. Keist, "Dynamic Processes", in *SystemC: From the Ground Up*, 2nd edition, USA: Springer Science+Business Media, 2009, pp. 89–98, ISBN: 978-0-387-69957-8. DOI: 10.1007/978-0-387-69958-5.

[25] IEEE Computer Society, "Elaboration and simulation semantics", in *1666-2011 IEEE Standard for SystemC Language Reference Manual*, New York, USA: IEEE, 2012, pp. 12–34, ISBN: 978-0-7381-6801-2 STD97162.

[26] C. Grimm, M. Barnasconi, A. Vachoux, and K. Einwich, *An Introduction to Modeling Embedded Analog/Mixed-Signal Systems Using SystemC AMS Extensions*, Open SystemC Initiative (OSCI), 2008.

[27] *Accellera Systems Initiative organization*. [Online]. Available: http://www.accellera.org/.

[28] M. Barnasconi, C. Grimm, M. Damm, K. Einwich, M.-M. Louërat, T. Maehne, F. Pêcheux, and A. Vachoux, *SystemC AMS extensions User's Guide*, Open SystemC Initiative (OSCI), 2010, 166 pp.

[29] *Fraunhofer SystemC-AMS. Fraunhofer IIS, Design Automation Division EAS*. [Online]. Available: http://systemc-ams.eas.iis.fraunhofer.de/.

[30] A. Vachoux, C. Grimm, and K. Einwich, "Towards Analog and Mixed-Signal SOC Design with SystemC-AMS", in *Proceeding of Second IEEE International Workshop on Electronic Design, Test and Applications, DELTA 2004.*, Perth, WA, Australia: IEEE, 2004, pp. 97–102, ISBN: 0-7695-2081-2. DOI: 10.1109/DELTA.2004.10008.

[31] K. Einwich, "Application of SystemC/SystemC-AMS for the Specification of Complex Wired Telecomunication Systems", in *Forum on specification and Design Languages (FDL)*, Lausanne, Switzerland, 2005, pp. 27–30.

[32] K. Einwich, P. Schwarz, C. Grimm, and K. Waldschmidt, "Mixed-Signal Extensions for SystemC", in *System Specification & Design Languages – Best of FDL'02*, E. Villar and J. Mermet, Eds., Springer US, 2003, pp. 19–28, ISBN: 978-1-4020-7414-1. DOI: 10.1007/0-306-48734-9_2.

[33] T. Uhle and K. Einwich, "A SystemC AMS Extension for the Simulation of Non-Linear Circuits", in *SOC Conference (SOCC), 2010 IEEE International*, Las Vegas, NV: IEEE, 2010, pp. 193–198, ISBN: 978-1-4244-6682-5. DOI: 10.1109/SOCC.2010.5784751.

[34] T. Maehne, "Efficient Modelling ans Simulation Methodology for the Design of Heterogeneous Mixed-Signal Systems on Chip", PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2011.

[35] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987. DOI: 10.1109/TC.1987.5009446.

[36] H. D. Patel and S. K. Shukla, "Deep vs. Shallow, Kernel vs. Language – What is better for Heterogeneous Modeling in SystemC?", FERMAT Lab., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, Tech. Rep. 2005-14, 2005, pp. 1–27.

[37] *CHESS: Center for Hybrid and Embedded Software Systems*. [Online]. Available: https://chess.eecs.berkeley.edu/.

[38] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment", *IEEE Computer*, vol. 36, pp. 45–52, 4 2003, ISSN: 0018-9162. DOI: 10.1109/MC.2003.1193228.

[39] A. Sangiovanni-Vincentelli, "Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design", in *Proceedings of the IEEE*, vol. 95, IEEE, 2007, pp. 467–506. DOI: 10.1109/JPROC.2006.890107.

[40] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu, "A Next-Generation Design Framework for Platform-Based Design", in *DVCon Conference*, vol. 152, 2007. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/228.html.

[41] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)", EECS Department, University of California, Berkeley, California, USA, Tech. Rep. UCB/EECS-2008-28, 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html.

[42] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorfer, S. Sachs, and Y. Xiong, "Taming Heterogeneity – The Ptolemy Approach", *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003, ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[43] E. A. Lee and H. Zheng, "Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems", in *Proceedings of the 7th ACM & Amp; IEEE International Conference on Embedded Software*, ser. EMSOFT '07, Salzburg, Austria: ACM, 2007, pp. 114–123, ISBN: 978-1-59593-825-1. DOI: 10.1145/1289927.1289949.

[44] E. A. Lee, "Heterogeneous Actor Modeling", in *Proceedings of the 9th ACM International Conference on Embedded Software*, ser. EMSOFT '11, Taipei, Taiwan: ACM, 2011, pp. 3–12, ISBN: 978-1-4503-0714-7. DOI: 10.1145/2038642.2038646.

[45] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)", EECS Department, University of California, Berkeley, California, USA, Tech. Rep. UCB/EECS-2008-37, 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-37.html.

[46] F. Herrera and E. Villar, "A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC", *ACM Transactions On Design Automation of Electronic Systems*, vol. 12, no. 3, 22:1–22:31, 2008, ISSN: 1084-4309. DOI: 10.1145/1255456.1255459.

[47] F. Herrera and E. Villar, *HetSC User Manual – Methodology for Specification of Heterogeneous Embedded Systems in SystemC*, GIM Group, TEISA Dpt., University of Cantabria, 2008. [Online]. Available: http://www.teisa.unican.es/HetSC/doc/HetSC_user_manual_2008_4_17.pdf.

[48] F. Herrera and E. Villar, *HetSC User Manual – Annexe B: Single MoC Specification*, GIM Group, TEISA Dpt., University of Cantabria, 2008. [Online]. Available: http://www.teisa.unican.es/HetSC/doc/HetSC_user_manual_ANNEXE_B_Single_MoC_2008_1_17.pdf.

[49] J. Zhu, I. Sander, and A. Jantsch, "HetMoC: Heterogeneous Modelling in SystemC", in *Proceedings of the 13th International Forum on Specification & Design Languages (FDL) 2010*, ECSI, Southampton, UK: IET, 2010. DOI: 10.1049/ic.2010.0139.

[50] S. H. Attarzadeh Niaki, M. K. Jakobsen, T. Sulonen, and I. Sander, "Formal Heterogeneous System Modeling with SystemC", in *Proceedings of the International Forum on Specification & Design Languages (FDL) 2012*, Vienna: IEEE, 2012, ISBN: 978-1-4673-1240-0.

[51] S. H. Attarzadeh Niaki, G. Silva Beserra, N. Andersen, M. Verdon, and I. Sander, "Heterogeneous System-Level Modeling for Small and Medium Enterprises", in *Proceedings of the 25th Symposium on Integrated Circuits and Systems Design (SBCCI) 2012*, Brasilia: IEEE, 2012, pp. 1–6, ISBN: 978-1-4673-2606-3. DOI: 10.1109/SBCCI.2012.6344450.

[52] S. H. Attarzadeh Niaki and I. Sander, "Co-Simulation of Embedded Systems in a Heterogeneous MoC-Based Modeling Framework", in *Proceedings of the 6th Symposium on Industrial Embedded Systems (SIES) 2011*, Vasteras: IEEE, 2011, pp. 238–247, ISBN: 978-1-61284-818-1. DOI: 10.1109/SIES.2011.5953667.

[53] T. Raudvere, I. Sander, A. K. Singh, D. Gurov, and A. Jantsch, "The ForSyDe Semantics", in *Proceedings of Swedish System-on-Chip Conference, 2002*, Sweden: IEEE, 2002, pp. 1–5.

[54] H. D. Patel and S. K. Shukla, "Towards a Heterogeneous Simulation Kernel for System-Level Models: a SystemC Kernel for Synchronous Data Flow Models", *Journal of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 8, pp. 1261–1271, 2005, ISSN: 0278-0070. DOI: 10.1109/TCAD.2005.850819.

[55] H. D. Patel and S. K. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling – A Framework for Multi-MoC Modeling & Simulation*. USA: Kluwer Academic Publishers, 2005, ISBN: 1-4020-8088-3.

[56] H. D. Patel, "HEMLOCK: HEterogeneous ModeL Of Computation Kernel for SystemC", Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2003.

[57] H. Al-Junaid and T. Kazmierski, "An Analogue and Mixed-Signal Extension to SystemC", in *IEE Proc. Circuits, Devices and Systems*, vol. 152, 6, 2005, pp. 682–690.

[58] C. Zhao and T. J. Kazmierski, "An extension to SystemC-A to support mixed-technology systems with distributed components", in *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE) 2011*, Grenoble, France: IEEE, 2011, ISBN: 978-1-61284-208-0. DOI: 10.1109/DATE.2011.5763205.

[59] H. Al-Junaid, T. Kazmierski, and L. Wang, "SystemC-A Modeling of an Automotive Seating Vibration Isolation System", in *Proceedings of the International Forum on Specification & Design Languages (FDL) 2006*, TU Darmstadt, Germany, 2006.

[60] C. Zhao and T. J. Kazmierski, “SystemC-A Modelling of Mixed-Technology Systems with Distributed Behaviour”, in *System Specification and Design Languages – Selected Contributions from FDL 2010*, USA: Springer New York, 2012, pp. 55–69, ISBN: 978-1-4614-1427-8. DOI: 10.1007/978-1-4614-1427-8_4.

[61] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow”, *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. DOI: 10.1109/PROC.1987.13876.

[62] C. Cassandras and S. Lafortune, “Petri Nets”, in *Introduction to Discrete Event Systems*, 2nd edition, USA: Springer Science+Business Media, LLC, 2008, pp. 223–267, ISBN: 978-0-387-33332-8. DOI: 10.1007/978-0-387-68612-7.

[63] K. Jensen and L. M. Kristensen, “CPN ML Programming”, in *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*, Springer-Verlag Berlin Heidelberg, 2009, pp. 43–77, ISBN: 978-3-642-00284-7. DOI: 10.1007/b95112.

[64] K. Jensen and L. M. Kristensen, “Timed Coloured Petri Nets”, in *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*, Springer-Verlag Berlin Heidelberg, 2009, pp. 231–255, ISBN: 978-3-642-00284-7. DOI: 10.1007/b95112.

[65] K. Jensen, S. Christensen, L. M. Kristensen, and M. Westergaard, *CPN Tools 4.0.0*, Architecture of Information Systems (AIS) Group, Eindhoven University of Technology, Netherlands, 2013. [Online]. Available: http://cpntools.org/.

[66] *CATRENE (CA701) H-INCEPTION. Heterogeneous Inception Project*, 2015. [Online]. Available: https://www-soc.lip6.fr/trac/hinception/.

[67] C. Ben Aoun, *Principes et réalisation d’un environnement de prototypage virtuel de systèmes hétérogènes composables*, 2015. [Online]. Available: https://edite-de-paris.fr/public/phd/html/10243011.

[68] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Creational Patterns”, in *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, pp. 81–136, ISBN: 0-201-63361-2.

[69] T. Maehne and A. Vachoux, “Bond Graph Support in SystemC AMS”, in *Proceedings of the 2012 10th International Conference on Bond Graph Modeling and Simulation (ICBGM’12)*, J. J. Granda and F. E. Cellier, Eds., ser. Simulation Series, SCS, AIAA, vol. 44, Genoa, Italy: Curran Associates, Inc., 2012, pp. 159–166, ISBN: 978-1-61839-985-4.

[70] T. Maehne, Z. Wang, L. Andrade, B. Vernay, C. Ben Aoun, J.-P. Chaput, M.-M. Louërat, F. Pêcheux, A. Krust, G. Schroepfer, M. Barnasconi, K. Einwich, F. Cenni, and O. Guillaume, “UVM-SystemC-AMS based Framework for the Correct by Construction Design of MEMS in their Real Heterogeneous Application Context,” in *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, Marseille, France, 2014, pp. 862–865. DOI: 10.1109/ICECS.2014.7050122.

[71] *Eigen C++ Template Library for Linear Algebra*. [Online]. Available: http://eigen.tuxfamily.org/.

APPENDIX

A Case Study Implementation

In this appendix we present the SystemC MDVP implementation of the vibration sensor model and its digital front end circuit presented in Section 7. The codes snippets, shown in Listings A.1-A.7, implement the set of TDF modules. Instances of these modules are used for defining the TDF model shown in Listing A.8.

```
1  class harmonic_sine_wavelets_source : public sca_tdf::sca_module {
2
3  public:
4      struct parameters { // Parameters of harmonic_sine_wavelets_source module
5          double offset; // Offset of the sine wave
6          double amplitude; // Amplitude of the sine wave
7          double f_0; // Base frequency of the sine wave
8          int n_period; // Number of periods of sine wave of base frequence per wavelet sequence
9          int n_harmonic; // Number of harmonics in the wavelet sequence.
10
11         parameters() // Initialize module parameters to sensible default values
12         : offset(-8.0e-6), amplitude(4.0e-6), f_0(2.0e3), n_period(8), n_harmonic(2) {}
13     };
14
15     sca_tdf::sca_out<double> out; // TDF output port
16
17     explicit harmonic_sine_wavelets_source ( sc_core::sc_module_name nm, const parameters& p = parameters())
18     : out("out"), offset_(p.offset), amplitude_(p.amplitude), f_0_(p.f_0),T_period_(static_cast<double>(p.n_period) / p.f_0),
19       n_harmonic_(p.n_harmonic) {}
20
21 protected:
22     void set_attributes() {
23         out.set_rate(1); // Output rate
24         out.set_delay(0); // Output delay
25     }
26
27     void processing() {
28         using namespace std;
29         double t = this->get_time().to_seconds();
```

```
30          double t_pos = fmod(t, T_period_);
31          int harmonic = static_cast<int>(floor(t / T_period_)) % (n_harmonic_ + 1);
32          double val = offset_;
33          val += amplitude_ * sin(2.0 * M_PI * pow(2.0, harmonic) * f_0_ * t_pos);
34          out.write(val);
35      }
36
37  private:
38      const double offset_; // Offset of the sine wave
39      const double amplitude_; // Amplitude of the sine wave
40      const double f_0_; // Base frequency of the sine wave
41      const double T_period_; // Period of a sine wavelet
42      const int n_harmonic_; // Number of harmonics in the wavelet sequence
43  };
```

Listing A.1: Source **SRC** Implemented by means of a TDF Module.

```
1   class vibration_sensor : public sca_tdf::sca_module {
2
3   public:
4       struct parameters { // Parameters of the vibration_sensor module
5           double k_trans; // Conversion factor from vibration velocity to output voltage.
6           parameters(): k_trans(1.0) {} // Initialize module parameters to sensible default values
7       };
8
9       sca_tdf::sca_in<double> in; // TDF input port (Displacement)
10      sca_tdf::sca_out<double> out; // TDF output port (Sensor output voltage)
11
12      explicit vibration_sensor (sc_core::sc_module_name nm, const parameters& p = parameters())
13      : in("in"), out("out"), k_trans_(p.k_trans), start_up_(true), in_last_(0.0), in_dot_(0.0) {}
14
15  protected:
16      void set_attributes() {
17          in.set_rate(1); // Input rate
18          in.set_delay(0); // Input delay
19          out.set_rate(1); // Output rate
20          out.set_delay(0); // Output delay
21      }
22
23      void processing() {
24          // Calculate velocity as 1st time derivative of displacement by evaluating Newton's difference quotient for the
25          // the current & the last sample
26          if (!start_up_) in_dot_ = (in.read() − in_last_) / in.get_timestep().to_seconds();
27          else start_up_ = false;
28          in_last_ = in.read();
29
30          out.write(k_trans_ * in_dot_); // Convert vibration velocity to output voltage via conversion factor k_trans
31      }
```

```
32 private:
33     const double k_trans_; // Conversion factor from vibration velocity to output voltage
34     bool start_up_; // Flag to mark first execution of processing()
35     double in_last_; // Last displacement read in prev. processing()
36     double in_dot_; // 1st time derivative of displacement, a.k.a. velocity
37 };
```

Listing A.2: Vibration Sensor **SENSOR** Implemented by means of a TDF Module.

```
1  class programmable_gain_amplifier : public sca_tdf::sca_module {
2
3  public:
4      struct parameters { // Parameters of the programmable_gain_amplifier module
5          double v_max; // Supply voltage limiting the output
6          parameters(): v_max(5.0) {} // Initialize module parameters to sensible default values
7      };
8
9      sca_tdf::sca_in<double> in; // TDF input port
10     sca_tdf::sca_de::sca_in<int> kin; // TDF input converter port
11     sca_tdf::sca_out<double> out; // TDF output port
12
13     explicit programmable_gain_amplifier (sc_core::sc_module_name nm, const parameters& p = parameters())
14     : in("in"), kin("kin"), out("out"), v_max_(p.v_max) {}
15
16 protected:
17     void set_attributes() {
18         in.set_rate(1); // Input rate
19         in.set_delay(0); // Input delay
20         kin.set_rate(1); // kin input rate
21         kin.set_delay(0); // kin input delay
22         out.set_rate(1); // output rate
23         out.set_delay(0); // output delay
24     }
25
26     void processing() {
27         double k = kin.read();
28         double val = std::pow(2.0, k) * in.read(); // Amplify input value.
29
30         // Test if output saturates due to amplified input value
31         if (val > v_max_) { out.write(v_max_); }
32         else if (val < -v_max_) { out.write(-v_max_); }
33         else { out.write(val); }
34     }
35
36 private:
37     const double v_max_; // Supply voltage limiting the output
38 };
```

Listing A.3: Programmable Gain Amplifier **PGA** Implemented by means of a TDF Module.

```
1  template<int NBits>
2  class ad_converter : public sca_tdf::sca_module {
3
4  public:
5      typedef sc_dt::sc_int<NBits> out_data_type;
6
7      struct parameters { // Parameters of the ad_converter module
8          double v_max; // Maximum input voltage
9          double tstep; // Module timestep
10         parameters(): v_max(5.0), tstep(0.000010) {} // Initialize module parameters to sensible default values
11     };
12
13     sca_tdf::sca_in<double> in; // TDF input port
14     sca_tdf::sca_out<out_data_type> out; // TDF output port
15
16     explicit ad_converter (sc_core::sc_module_name nm, const parameters& p = parameters())
17     : in("in"), out("out"), v_max_(p.v_max), tstep_(p.tstep) {}
18
19 protected:
20     void set_attributes() {
21         this->set_timestep(tstep_, sc_core::SC_SEC);
22         in.set_rate(10); // Input rate
23         in.set_delay(0); // Input delay
24         out.set_rate(1); // Output rate
25         out.set_delay(0); // Output delay
26     }
27
28     void processing() {
29         using namespace std;
30         double v_in = in.read();
31
32         //! Take into account saturation if input voltage range is exceeded.
33         if (v_in < -v_max_) { out.write(-((1 << (NBits - 1)) - 1)); }
34         else if (v_in > v_max_) { out.write((1 << (NBits - 1)) - 1); }
35         else {
36             sc_dt::sc_int<NBits>
37             q_v_in = lround((v_in / v_max_) * ((1 << (NBits - 1)) - 1));
38             out.write(q_v_in);
39         }
40     }
41
42 private:
43     const double v_max_; // Maximum input voltage
44     double tstep_; // Module timestep
45 };
```

Listing A.4: Analog to Digital Converter **ADC** Implemented by means of a TDF Module.

```
1  template<int NBits>
2  class tdf2de : public sca_tdf::sca_module {
3
4  public:
5      typedef sc_dt::sc_int<NBits> data_type;
6      sca_tdf::sca_in<data_type> in; // TDF input port
7      sca_tdf::sca_de::sca_out<data_type> out; // TDF output converter port
8
9      tdf2de<NBits>::tdf2de(sc_core::sc_module_name nm)
10     : in("in"), out("out") {}
11
12 protected:
13     void set_attributes() {
14         in.set_rate(1); // Input rate
15         in.set_delay(0); // Input delay
16         out.set_rate(1); // Output rate
17         out.set_delay(0); // Output delay
18     }
19
20     void processing() {
21         out.write(in.read());
22     }
23 };
```

Listing A.5: TDF to DE Converter **TDF2DE** Implemented by means of a TDF Module.

```
1  template<int NBits>
2  class abs_amplitude_averager : public sca_tdf::sca_module {
3
4  public:
5      typedef sc_dt::sc_int<NBits> data_type;
6
7      struct parameters { // Parameters of the abs_amplitude_averager module
8          long ns_0; // Initial Number of averaged samples
9          parameters(): ns_0(64) {} // Initialize module parameters to sensible default values
10     };
11
12     sca_tdf::sca_in<data_type> in; // TDF input port
13     sca_tdf::sca_de::sca_out<bool> clk; // TDF output converter port
14     sca_tdf::sca_de::sca_out<data_type> amp; // TDF output converter port
15
16     explicit abs_amplitude_averager (sc_core::sc_module_name nm, const parameters& p = parameters())
17     : in("in"), clk("clk"), amp("amp"), ns_(p.ns_0) {}
18
19 protected:
20     void set_attributes() {
21         in.set_rate(ns_); // Input rate
22         in.set_delay(0); // Input delay
```

```
23          clk.set_rate(2); // Output clk rate
24          clk.set_delay(0); // Output clk delay
25          amp.set_rate(1); // Output amp rate
26          amp.set_delay(0); // Output amp delay
27      }
28
29      void processing() {
30          // Generate clock signal
31          clk.write(true, 0);
32          clk.write(false, 1);
33
34          // Calculate and output average of absolute amplitudes
35          long sum = 0;
36          for (long i = 0; i < ns_; ++i) {
37              sum += std::labs(in.read(i));
38          } long avg = sum / ns_;
39
40          amp.write(avg); // Write average of absolute amplitudes in the TDF output converter port
41      }
42
43  private:
44      long ns_; // Number of averaged samples
45  };
```

Listing A.6: Amplitude Estimator **AAVG** Implemented by means of a TDF Module.

```
1   template<int NBits>
2   class gain_controller : public sc_core::sc_module {
3
4   public:
5       typedef sc_dt::sc_int<NBits> data_type;
6
7       struct parameters { // Parameters of the digital gain controller module
8           int threshold_min; // Low threshold for amplitude to increase gain
9           int threshold_max; // High threshold for amplitude to lower gain
10          int k_0; // Initial gain power
11          int k_min; // Minimum gain power
12          int k_max; // Maximum gain power
13
14          parameters(): threshold_min(0.2 * ((1 << (NBits - 1)) - 1)), threshold_max(0.6 * ((1 << (NBits - 1)) - 1)), k_0(8),
15          k_min(0), k_max(16) {}
16      };
17
18      sc_core::sc_in<bool> clk; // Input DE port
19      sc_core::sc_in<sc_dt::sc_int<NBits> > in; // Input DE port
20      sc_core::sc_out<int> out; // Output DE port
21
22      SC_HAS_PROCESS(gain_controller);
```

```
23      explicit gain_controller(sc_core::sc_module_name nm, const parameters& p = parameters())
24      : clk("clk"), in("in"), out("out"), threshold_min_(p.threshold_min), threshold_max_(p.threshold_max),
25        k_min_(p.k_min), k_max_(p.k_max), state_(keep_gain), k_(p.k_0) {
26          SC_METHOD(adapt_gain);
27          sensitive << clk.pos();
28      }
29
30  private:
31      const int threshold_min_; // Low threshold for amplitude to increase gain
32      const int threshold_max_; // High threshold for amplitude to lower gain
33      const int k_min_; // Minimum gain power
34      const int k_max_; // Maximum gain power
35      enum state_type {keep_gain, increase_gain, decrease_gain}; // Possible states of the gain controller FSM
36      state_type state_; // Current state
37      int k_; // Current gain power
38
39      void adapt_gain() {
40          // Perform actions and state transitions based on the current state
41          switch (state_) {
42              case keep_gain:
43                  if (in.read() < threshold_min_) {
44                      state_ = increase_gain;
45                      ++k_;
46                  }
47                  else if (in.read() >= threshold_max_) {
48                      state_ = decrease_gain;
49                      --k_;
50                  }
51                  break;
52              case increase_gain:
53                  if (in.read() < threshold_max_) {
54                      ++k_;
55                  } else {
56                      state_ = decrease_gain;
57                      --k_;
58                  }
59                  break;
60              case decrease_gain:
61                  if (in.read() < threshold_max_) {
62                      state_ = keep_gain;
63                  } else {
64                      --k_;
65                  }
66                  break;
67              default:
68                  SC_REPORT_ERROR("gain_controller", "Unexpected_state.");
69          }
70
```

```
71          // Limit and set new gain.
72          if (k_ < k_min_) { k_ = k_min_; }
73          if (k_ > k_max_) { k_ = k_max_; }
74          out.write(k_);
75      }
```

Listing A.7: Gain Controller **CTRL** Implemented by means of a DE Module.

```
1   int sc_main(int argc, char* argv[]) {
2       using namespace sc_core;
3
4       // Simulation conditions
5       double init_ADC_Tm = 0.000010; // Timestep
6       const sc_time t_stop(25.00, SC_MS); // Simulation stop time.
7
8       // Source parameters
9       harmonic_sine_wavelets_source::parameters vib_src_params;
10      vib_src_params.offset = -8.0e-6;
11      vib_src_params.amplitude = 4.0e-6;
12      vib_src_params.f_0 = 2.0e3;
13      vib_src_params.n_period = 8;
14      vib_src_params.n_harmonic = 2;
15
16      // Vibration sensor parameters
17      vibration_sensor::parameters vib_sensor_params;
18      vib_sensor_params.k_trans = 1.0;
19
20      // Programmable gain amplifier parameters
21      programmable_gain_amplifier::parameters pga_params;
22      pga_params.v_max = 5.0;
23
24      // AD converter parameters
25      const int NBitsADC = 5; // Resolution of the ADC
26      ad_converter<NBitsADC>::parameters adc_params;
27      adc_params.v_max = 5.0;
28      adc_params.tstep = init_ADC_Tm;
29
30      // Absolute amplitude averager parameters
31      abs_amplitude_averager<NBitsADC>::parameters abs_params;
32      abs_params.ns_0 = 64;
33
34      // Gain controller parameters
35      gain_controller<NBitsADC>::parameters gain_ctrl_params;
36      gain_ctrl_params.threshold_min = 0.2 * ((1 << (NBitsADC - 1)) - 1);
37      gain_ctrl_params.threshold_max = 0.6 * ((1 << (NBitsADC - 1)) - 1);
38      gain_ctrl_params.k_0 = 8;
39      gain_ctrl_params.k_min = 0;
40      gain_ctrl_params.k_max = 16;
```

```
41      // TDF signals
42      sca_tdf::sca_signal<double> x_sig("x_sig");
43      sca_tdf::sca_signal<double> v_sig("v_sig");
44      sca_tdf::sca_signal<double> vamp_sig("vamp_sig");
45      sca_tdf::sca_signal<sc_dt::sc_int<NBitsADC> > adc_sig("adc_sig");
46      // DE signals
47      sc_core::sc_signal<sc_dt::sc_int<NBitsADC> > out_sig("out_sig");
48      sc_core::sc_signal<int> k_sig("k_sig");
49      sc_core::sc_signal<sc_dt::sc_int<NBitsADC> > amp_sig("amp_sig");
50      sc_core::sc_signal<bool> clk_sig("clk_sig");
51
52      // Mechanical vibration source instance
53      harmonic_sine_wavelets_source
54      vib_src("SRC", vib_src_params);
55      vib_src.out(x_sig);
56
57      // Vibration sensor instance with displacement input and velocity proportional voltage output
58      vibration_sensor vib_sensor("SENSOR", vib_sensor_params);
59      vib_sensor.in(x_sig);
60      vib_sensor.out(v_sig);
61
62      // Programable gain amplifier instance
63      programmable_gain_amplifier pga("PGA", pga_params);
64      pga.in(v_sig);
65      pga.kin(k_sig);
66      pga.out(vamp_sig);
67
68      // AD converter instance
69      ad_converter<NBitsADC> adc("ADC", adc_params);
70      adc.in(vamp_sig);
71      adc.out(adc_sig);
72
73      // TDF2DE converter instance
74      tdf2de<NBitsADC> tdf2de("TDF2DE");
75      tdf2de.in(adc_sig);
76      tdf2de.out(out_sig);
77
78      // Absolute amplitude averager instance
79      abs_amplitude_averager<NBitsADC> abs("AAVG", abs_params);
80      abs.in(adc_sig);
81      abs.clk(clk_sig);
82      abs.amp(amp_sig);
83
84      // Gain controller instance
85      gain_controller<NBitsADC> gain_ctrl("CTRL", gain_ctrl_params);
86      gain_ctrl.clk(clk_sig);
87      gain_ctrl.in(amp_sig);
88      gain_ctrl.out(k_sig);
```

```
89      // Simulation
90      try {
91          sc_start(t_stop);
92      } catch (const std::exception& e) {
93          std::cerr << e.what() << std::endl;
94      } sc_stop();
95
96      return sc_report_handler::get_count(SC_ERROR);
97  }
```

Listing A.8: Implementation of the Modules' Composition shown in Figure 7.1.

APPENDIX

B Publications

- **L. Andrade**, T. Maehne, A. Vachoux, C. Ben Aoun, F. Pêcheux, M.-M. Louërat, *Pre-Simulation Symbolic Analysis of Synchronization Issues between Discrete Event and Timed Data Flow Models of Computation*, DATE 2015, Grenoble, France.
- **L. Andrade**, C. Ben Aoun, B. Vernay, T. Maehne, F. Pêcheux, M.-M. Louërat, *Understanding the Heterogeneous Hardware: Do not forget the interconnection!*, DUHDe at DATE 2015, Grenoble, France.
- C. Ben Aoun, **L. Andrade**, T. Maehne, F. Pêcheux, M.-M. Louërat, A. Vachoux, *Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach*, SAMOS 2015, Samos, Greece.
- C. Ben Aoun, **L. Andrade**, T. Maehne, F. Pêcheux, M.-M. Louërat, A. Vachoux, *Generic EDA-Standard Based Elaboration Scheme for the Efficient Monolithic Simulation of Heterogeneous Systems*, Work-in-progress session at DAC 2015, San Francisco, United States.
- T. Maehne, Zh. Wang, B. Vernay, **L. Andrade**, C. Ben Aoun, J.-P. Chaput, M.-M. Louërat, F. Pêcheux, A. Krust, G. Schroepfer, M. Barnasconi, K. Einwich, F. Cenni, O. Guillaume, *UVM-SystemC-AMS based Framework for the Correct by Construction Design of MEMS in their Real Heterogeneous Application Context*, ICECS 2014, Marseille, France.
- **L. Andrade**, T. Maehne, M.-M. Louërat, F. Pêcheux, *Time Step Control and Threshold Crossing Detection in SystemC AMS 2.0*, GDR SoC-SIP du CNRS 2013, Lyon, France.

APPENDIX

C

Résumé en Français

Contents

C.1. Introduction (chapitre 1)

Ce chapitre est un résumé étendu de la thèse en français. Chaque section correspond à un chapitre de la thèse.

C.1.1. Le contexte

Cette thèse est contemporaine de la révolution de l'Internet des Choses. Ces "choses" sont des systèmes complexes qui communiquent entre eux par radio-fréquence et interagissent sur le monde extérieur grâce à des capteurs et des actionneurs. Ces objets sont complexes à plusieurs titres : d'une part parce que le coeur numérique embarqué interagit avec le monde physique au travers de composants qui relèvent de plusieurs disciplines (mécanique, thermique, chimie), et d'autre part parce que la modélisation du comportement de ces systèmes met en jeu plusieurs représentations du temps : continu, échantillonné, à événement discrets, flot de données. Il s'agit donc d'une hétérogénéité qui concerne les données et le temps, qui nécessite de synchroniser les différentes échelles de temps et les données associées.

C.1.2. Contribution et organisation de la thèse

La thèse comporte 8 chapitres et 3 annexes.

Le chapitre 1 est une introduction qui résume l'objectif de cette thèse et les principales contributions. L'objectif principal de ce travail est de comprendre comment simuler un système hétérogène fondé sur le standard SystemC [14] et son extension AMS [13]. Le chapitre 2 explique plus en détail les problèmes posés par la modélisation et la simulation d'un système hétérogène. Le chapitre 3 donne l'état de l'art sur le sujet. Le chapitre 4 explique en détail les problèmes de causalité qui apparaissent au cours de la simulation d'un système modélisé suivant les événements discrets (DE) pour une part et suivant le temps échantillonné (DT) d'autre part. Ce chapitre propose une formalisation de ce problème à l'aide des réseaux de Petri colorés [17] et une solution systématique pour détecter les problèmes de synchronisation DE/DT avant l'exécution effective de la simulation. Le deuxième objectif de cette thèse est de proposer un mécanisme générique pour ajouter des modèles de calcul à SystemC, permettant d'étendre les domaines temporels et les disciplines pouvant être simulées. Ce mécanisme est expliqué au chapitre 5. Ces propositions sont mises en oeuvre au chapitre 6 qui décrit l'intégration au simulateur SystemC [14] d'un nouveau modèle de calcul TDF conforme au standard AMS [13]. L'efficacité de cette approche est illustrée par la simulation d'un modèle décrivant le comportement d'un capteur de vibration au chapitre 7. La conclusion et les perpectives de la thèse sont données au chapitre 8.

C.2. Motivation et définition de la problématique (chapitre 2)

C.2.1. Introduction

La modélisation et la simulation d'un système hétérogène sont des étapes importantes dans la conception d'un système sur puce (SoC). Ces circuits intègrent sur le même substrat silicium des fonctions numériques (processeurs, mémoires, bus ou réseau d'interconnexion, timers), des fonctions radio-fréquence (émetteur, récepteur), des fonctions analogiques en bande de base (convertisseurs analogiques-numériques, convertisseurs DC-DC, régulateur de tension) mais également des cap-

teurs et des actionneurs comportant des éléments mécaniques ou d'autres disciplines physiques. Ces circuits intègrent donc plusieurs disciplines dont les comportements sont représentés par des modèles de temps différents : événements, temps continu, temps échantillonné. Un exemple est présenté Figure C.1.

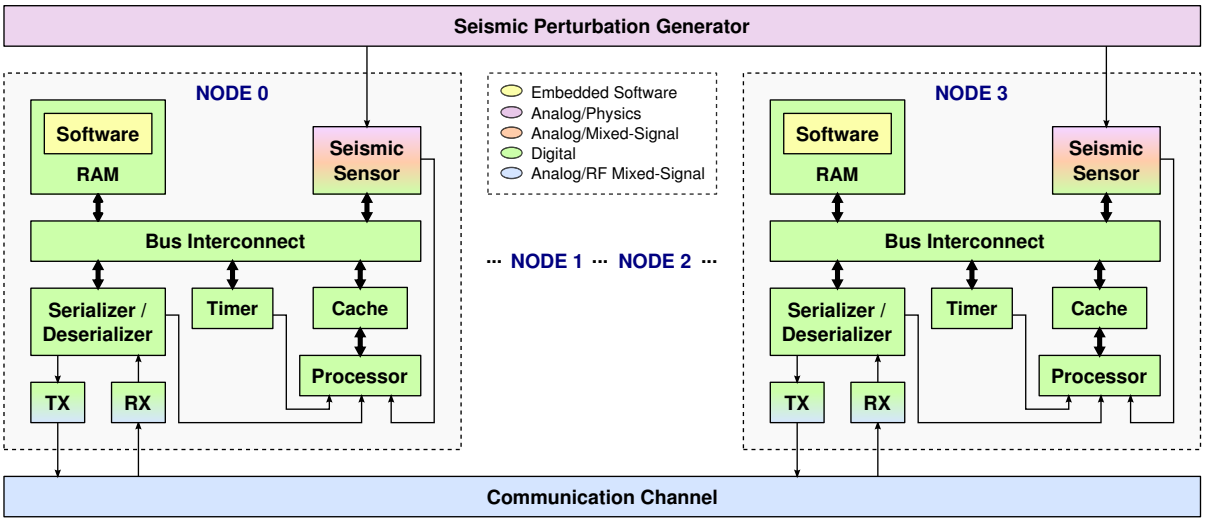

Figure C.1: Mise en oeuvre d'un réseau de capteurs sans fil (WSN) pour calculer l'épicentre d'une perturbation sismique (adapté de [18]).

Comme la complexité de ces SoC ne cesse de croître, le temps de mise sur le marché d'un nouveau produit nécessite de développer en parallèle le matériel et le logiciel, ainsi que la synthèse et la vérification. Les diverses performances (fonctionnalité, architecture, réalisation physique, vitesse, consommation) doivent être traitées et analysées pour l'ensemble du SoC [19]. Pour répondre à cette attente des concepteurs, le consortium Accellera Systems Initiative (initialement Open SystemC Initiative) a standardisé les extensions AMS de SystemC [13]. Mais le standard, fort utile au demeurant, laisse plusieurs points difficiles à la charge des concepteurs : le premier est la définition d'un mécanisme robuste pour assurer la synchronisation entre le monde à événements discrets (DE, SystemC) et celui du temps échantillonné (DT de SystemC AMS), le deuxième est la définition d'un mécanisme générique pour ajouter au simulateur SystemC-AMS un nouveau moteur de calcul pour modéliser et simuler une nouvelle discipline.

Les caractéristiques de SystemC et de SystemC AMS, utiles pour expliquer les contributions de la thèse à la modélisation et la simulation de systèmes hétérogènes, sont expliquées dans la suite de cette section.

C.2.2. Principes de simulation à événements discrets (DE) du standard SystemC

SystemC [20]–[22] est un langage de modélisation qui permet de représenter le matériel "numérique" et le logiciel embarqué sur un SOC, sous la forme d'une bibliothèque C++. Il est considéré à la fois comme un langage de spécifications (exécutables) au niveau système et comme un langage de description de matériel, puis qu'il permet d'aller jusqu'au niveau RTL. SystemC met en œuvre plusieurs concepts qui sont bien adaptés à la modélisation du matériel : une horloge globale, qui est gérée par un moteur de simulation, les processus concurrents qui sont gérés par un ordonnanceur, des types de données de

type entiers ou bits, la hiérarchie, la communication entre modules à travers des interfaces, ports et canaux.

Le moteur de simulation de SystemC [25] offre les structures fondamentales pour les phases d'élaboration et de simulation des modèles. L'élaboration consiste à créer les structures de données nécessaires à la simulation telles que la hiérarchie des modules, l'instanciation des processus, les connexions entre ports par les canaux et la résolution temporelle minimale. La simulation lance l'ordonnancement des processus et efface les structures de données créées lors de la phase d'élaboration. La sémantique d'élaboration et de simulation du standard SystemC est illustrée par la Figure C.2.

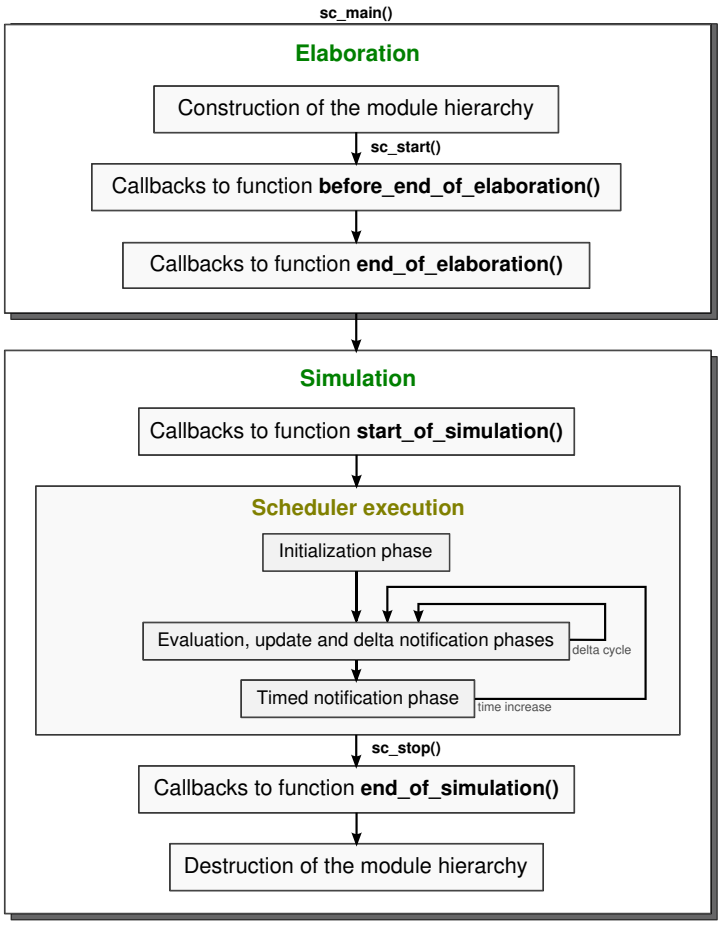

Figure C.2: Sémantique d'élaboration et de simulation du standard SystemC.

L'ordonnancement est au coeur de SystemC puisqu'il contrôle l'ordre dans lequel sont exécutés les processus. Notons qu'en plus des phases d'exécution des processus et de l'effacement des structures, il apparait 4 fonctions de rappel :

- before_end_of_elaboration(): qui permet d'agir sur la hiérarchie des modules,
- end_of_elaboration(): qui permet de faire des tests sur la structure hiérarchique,
- start_of_simulation(): qui permet d'ouvrir des fichiers et de créer des processus dynamiques,
- end_of_simulation(): qui permet de fermer de fichiers.

Ces 4 fonctions de rappel ont un rôle fondamental car elles permettent de proposer des extensions au langage de modélisation et de simulation SystemC. Les fonctions end_of_elaboration() et start_of_simulation() seront utilisées par le simulateur multi-discipline proposé dans cette thèse.

C.2.3. La standardisation du langage SystemC AMS

Dans cet esprit, les extensions AMS de SystemC ont été proposées dans le but d'offrir un simulateur de systèmes mixtes analogiques-numériques, à temps continu pour répondre aux demandes de nombreuses applications industrielles dans le domaine des télécommunications, de l'automobile, et de l'industrie des semi-conducteurs [26].

Ces extensions ont été définies comme une bibliothèque de fonctions C++, qui peut traiter aussi bien la simulation à événement discret (DE), que celle en temps discret ou échantillonné (alors appelé TDF) et celle en temps continu. Le premier *Language Reference Manual (LRM)* a été standardisé par l'OSCI en 2010 [13], accompagné par un guide de l'utilisateur [28]. Actuellement il n'existe qu'une seule preuve de concept [16] développée par l'institut Fraunhofer de Dresde [29].

Puisque les SoCs d'aujourd'hui sont hétérogènes, leur modélisation et simulation doivent savoir combiner plusieurs types de modèles de calcul (MoC). C'est pourquoi le langage SystemC AMS est hiérarchique [30], à plusieurs niveaux présentés à la Figure C.3, fondés sur SystemC.

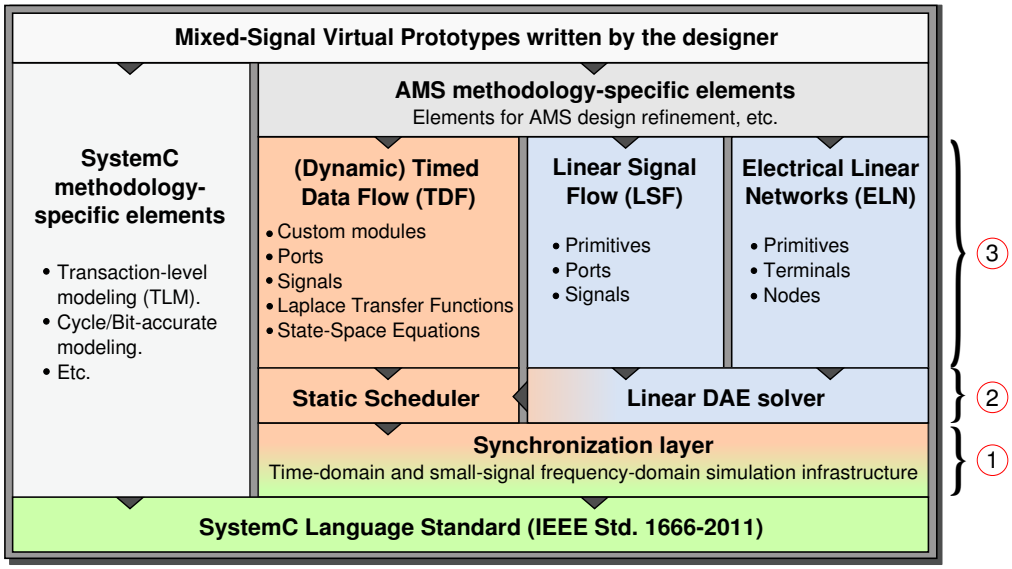

Figure C.3: Le standard du langage SystemC AMS (adapté de [28]).

Examinons quels sont ces niveaux :

- Le premier constitue la couche de *synchronisation*, notée par (1). Elle interagit directement avec SystemC et est responsable de réaliser un ordonnancement pour la simulation, qui consiste à déterminer les dates de synchronisation entre la simulation numérique et la simulation analogique, à appeler les solveurs et à établir la communication entre les solveurs. En SystemC AMS, un **solveur** doit calculer la solution à l'aide d'une formule mathématique, mais également définir les phases d'élaboration et de simulation spécifiques à ce MoC.

- Le deuxième constitue la couche des *solveurs*, notée par (2). Elle interagit avec la couche de synchronisation, calcule le comportement des sous-systèmes analogiques à l'aide d'algorithmes

dédiées à chaque MoC.

- Le troisième est la couche *view layer*, notée par (3). Elle offre au concepteur de SoC, les interfaces pour décrire les modèles (comportement ou netlist). Elle fournit les méthodes pour créer les structures qui seront utilisées par les solveurs pendant la simulation.

C.2.4. Le modèle de calcul (MoC) à flot de données échantillonné en temps (TDF) du standard SystemC AMS

Le modèle de calcul (MoC) à flot de données échantillonné en temps (TDF) du standard SystemC AMS [28] repose sur le formalisme des Synchronous Data Flow (SDF) [35]. Le comportement d'un bloc est décrit en considérant une donnée comme un signal, dont les valeurs sont échantillonnées au cours du temps selon un pas de temps constant. Le MoC TDF conserve les propriétés d'un modèle SDF : l'existence d'un ordonnancement statique et l'exécution périodique auxquelles il ajoute la notion de temporalité pour interagir avec d'autres MoC dépendant du temps.

Un modèle TDF (Figure C.4), est composé typiquement de modules TDF (notés (1)) interconnectés par des signaux TDF (notés (2)). Les connexions des modules aux signaux se font par des ports (notés (3)). Lorsqu'un module TDF doit interagir avec un module DE (pour le contrôle par exemple), les modules DE (SystemC, notés (4)) sont connectés aux modules TDF par des signaux SystemC (notés (6)). Les modules SystemC sont connectés aux modules TDF par des ports de conversion d'entrée TDF (notés (7)). Les connexions depuis un module TDF vers un module SystemC, typiquement pour le traitement numérique des signaux, sont réalisées par des ports de conversion de sortie TDF (notés (8)). L'ensemble des modules TDF (noté (9)) s'appelle un cluster. Un cluster est caractérisé par un ordonnancement statique de ses modules pour exécuter la simulation. La Figure C.4, illustre deux clusters : le premier, composé des modules **A** et **B** et le second des modules **C** et **D**.

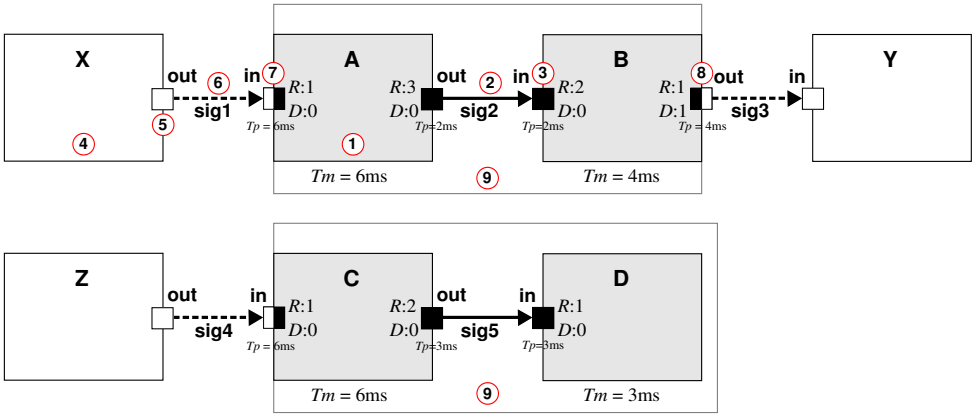

Figure C.4: Exemple simple d'un modèle TDF à taux d'échantillonnages multiples, cas de 2 clusters, 4 modules TDF et 2 signaux TDF. L'interaction avec le domaine temporel à événement discret (DE) met en œuvre des ports de conversion.

Un module M est décrit par :

- **un attribut** *Tm*. Il s'agit de la période d'exécution de la fonction processing() du module.
- **une fonction** processing(). Il s'agit d'une fonction, au sens mathématique du terme, qui dépend des entrées du module et éventuellement de son état, et calcule les sorties.

Un port est quant à lui décrit par 3 attributs :

- **le pas de temps** *Tp*. Il s'agit de la période à laquelle un port peut lire ou écrire des données.
- **le taux d'échantillonnage** *R*. Il s'agit du nombre de données qui peuvent être lues ou écrites par un port durant une prériode *Tp*.
- **le retard** *D*. Il s'agit du nombre de données qui doivent être disponibles à l'initialisation de la simulation, pour respecter l'ordonnancement du cluster. Ce nombre vaut zéro par défaut. Il peut être non nul dans le cas de taux d'échantillonnage multiple en interaction avec un module SystemC ou en cas d'un ensemble de modules TDF reliés en boucle.

Pour être conforme au schéma de simulation de SystemC présenté à la Figure C.2, l'exécution d'un modèle AMS/TDF suit un mécanisme à deux phases : la phase d'*élaboration TDF*, exécutée dans le contexte de la fonction de rappel end_of_elaboration() de SystemC; et la phase de *simulation TDF*, exécutée dans le contexte de la fonction de rappel start_of_simulation() de SystemC au premier *delta cycle* de l'ordonnanceur SystemC. La Figure C.5 illustre ces phases.

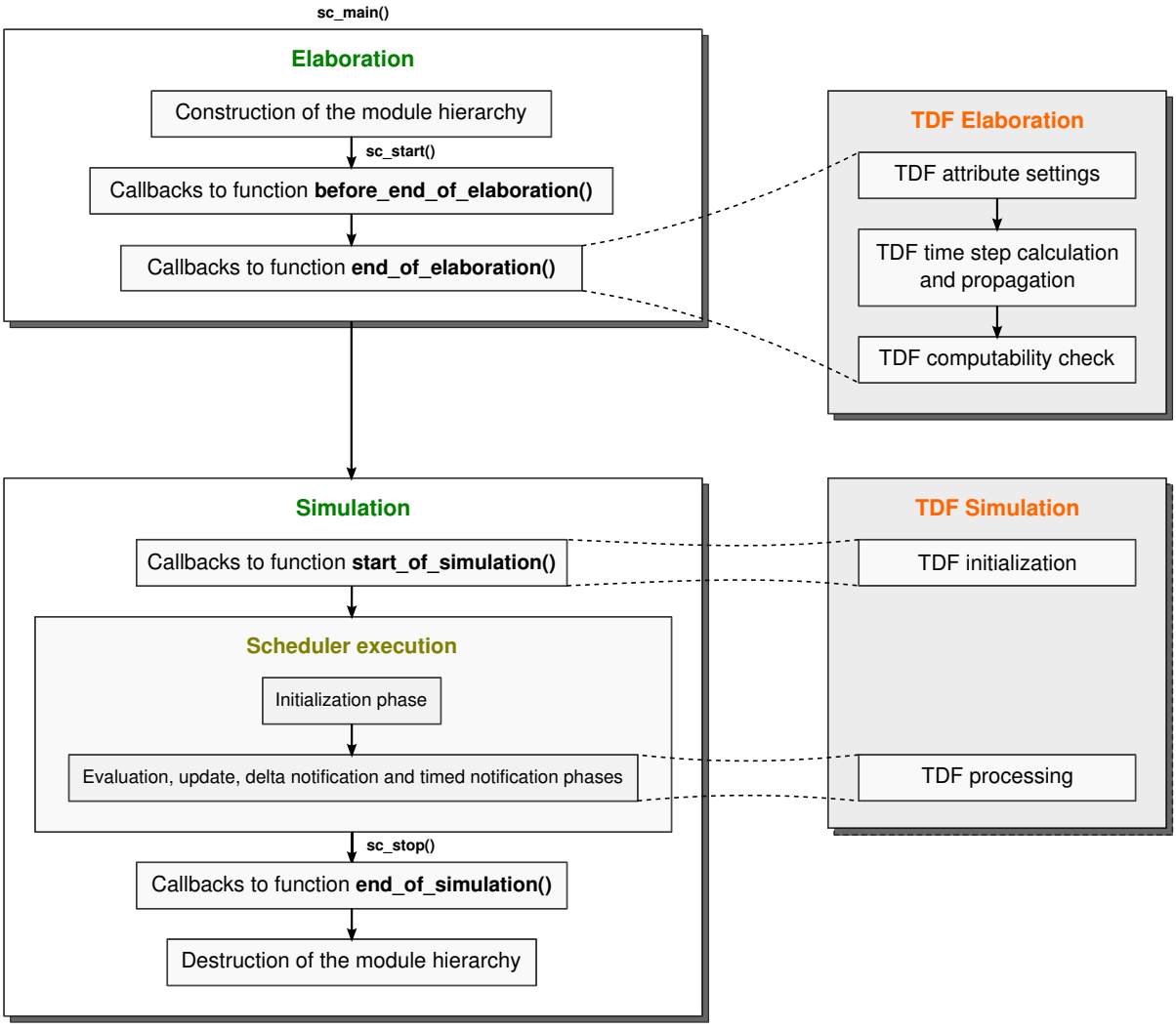

Figure C.5: Sémantique d'élaboration et de simulation SystemC-AMS.

Pendant la phase d'élaboration de la simulation AMS/TDF, les modules sont regroupés en cluster. Pour chaque cluster, le simulateur établit un ordonnancement statique. Cette étape commence par vérifier qu'un ordonnancement existe. Ceci suppose de vérifier à l'intérieur d'un cluster, la cohérence des périodes d'activation de chacun des modules et de leurs ports qui doivent satisfaire la condition C.1:

$$Tm = Tp * R \tag{C.1}$$

et que les ports reliés par un même signal TDF vérifient l'équation C.2 :

$$q_{M_i} * R_i = q_{M_j} * R_j \tag{C.2}$$

où q_{M_i} et q_{M_j} sont deux entiers qui déterminent le nombre d'activations respectives de M_i et M_j et permettent d'établir l'ordonnancement du cluster contenant ces modules.

Pendant la phase de simulation, les fonctions d'initialisation de chacun des modules sont exécutées dans un ordre quelconque, puis les fonctions de calcul du comportement sont exécutées suivant l'ordonnancement établi.

C.2.5. Le problème

La preuve de concept PoC [16] qui a été développée à l'institut Fraunhofer de Dresde [29] ne répond pas complètement au problème de simulation d'un système hétérogène. Nous avons identifié dans cette preuve de concept (PoC) trois points susceptibles d'être améliorés :

- L'ordonnancement des clusters TDF, dans la PoC SystemC-AMS, est fait de manière interne à chaque cluster, en dehors du contexte de simulation SystemC. Cela peut entrainer des erreurs de causalité qui ne peuvent être détectées que lors de la simulation. Nous proposons de formaliser le mécanisme de synchronisation DE/TDF et de l'intégrer à l'ordonnancement d'un cluster TDF.

- Il est très difficile d'intégrer un nouveau modèle de calcul à la PoC SystemC-AMS et cela nécessite toujours d'intervenir au cœur du noyau de simulation SystemC-AMS/TDF. Nous proposons d'établir un mécanisme générique pour ajouter un nouveau modèle de calcul au noyau SystemC.

- Une autre raison limite l'ajout d'un nouveau modèle de calcul : la couche de synchronisation avec SystemC de la PoC actuelle repose nécessairement sur la synchronisation entre DE et TDF. Cela signifie que la synchronisation d'un nouveau modèle de calcul doit respecter la sémantique de synchronisation de TDF. Bien que le standard TDF soit un type de modélisation à privilégier pour représenter les systèmes hétérogènes, nous souhaitons autoriser d'autres types de modélisation et donc d'interaction avec SystemC. Dans ce but, nous proposons un mécanisme de synchronisation générique avec le noyau de simulation de SystemC directement.

C.3. État de l'art (chapitre 3)

Le chapitre 3 présente un état de l'art des approches rencontrées pour modéliser et simuler des systèmes multi-disciplines, ou cyber-physiques (CPS). Nous avons distingué trois classes d'approches :

- Les environnements de modélisation et simulation qui s'appuient sur les méta-modèles et les langages de programmation haut niveau. Il s'agit de Metropolis [38], [39], Metro II [40] et Ptolemy II [41]–[44]. Ils sont présentés par le Tableau C.1.

- Les environnements de modélisation et simulation fondés sur SystemC. Il s'agit de HetSC [46], [47], HetMoC [49] et ForSyDe [50]–[52]. Ils sont présentés par le Tableau C.2.

- Les environnements de modélisation et simulation qui ont proposé des extensions au noyau de simulation DE de SystemC. Il s'agit de SystemC-H [54]–[56] et SystemC-A [57], [58].

Les tableaux C.1 et C.2 résument et comparent les différentes approches rencontrées dans les deux premiers points.

Pour effectuer la comparaison, nous avons examiné à quel degré le caractère hétérogène du système modélisé et simulé, peut être envisagé. Pour cet examen, nous nous sommes appuyés sur la définition donnée par [36] qui distingue clairement l'hétérogénéité *superficielle* (exprimée à travers le langage de modélisation, mais le modèle de calcul sous-jacent est unique) de l'hétérogénéité *profonde* (où différentes disciplines peuvent utiliser des moteurs de calcul différents).

La dernière classe d'environnements que nous avons examinés concerne les noyaux de simulation qui ont modifié le standard SystemC. Il s'agit de SystemC-H [55] et SystemC-A [59], [60]. Ces environnements permettent d'ajouter de nouveaux moteurs de calculs au noyau de simulation de SystemC, ce qui ouvre de nouvelles perspectives à la modélisation et simulation de systèmes hétérogènes. D'une part, SystemC-H a l'avantage de proposer une relation maître-esclave entre les modèles de calcul et SystemC-A de traiter le domaine analogique en temps continu. D'autre part ils reposent sur des choix stratégiques qui ne nous semblent pas convenir à l'heure actuelle :

- l'absence de représentation hiérarchique,

- les classes SystemC ne sont pas utilisées comme fondement de nouveaux composants,

- le noyau de simulation de SystemC est modifié,

- on ne trouve pas de mécanisme de synchronisation avec des domaines autres que le domaine des événements discrets de SystemC.

C'est pourquoi, après avoir examiné les approches existantes, nous souhaitons définir un environnement de modélisation et simulation qui possède les caractéristiques suivantes :

- la modélisation hiérarchique,

- l'hétérogénéité profonde mettant en œuvre plusieurs noyaux de simulation,

- les modèles de calcul liés par des relations maître-esclave,

- des composants assurant la synchronisation du temps et des données, sans être à la charge du concepteur du SoC,

- la sémantique de synchronisation définie de façon formelle,

- le noyau de simulation de SystemC conservé et les nouvelles classes de composants héritant des propriétés des classes de SystemC.

Environnement	**Metropolis**	**Metro II**	**Ptolemy II**
La modélisation hiérarchique	Pas autorisée.	Un niveau hiérarchique est possible via l'import d'un IP existant.	Plusieurs niveaux autorisés via les *acteurs composites*.
Séparation entre le calcul et la communication	- Le calcul s'appuie sur les *processus*. - La communication s'appuie sur les *ports*, *interfaces* et *mediums*.	- Le calcul s'appuie sur les *composants*. - La communication s'appuie sur les *ports* et *connexions*.	- Le calcul s'appuie sur les *acteurs*. - La communication s'appuie sur les *ports* et *canaux*.
Degré d'hétérogénéité	Langage.	Langage.	Noyau.
La synchronisation	- La synchronisation du temps s'appuie sur les *contraintes* et géré par les *quantity managers*. - La synchronisation des données est gérée par les *mediums*.	- La synchronisation du temps s'appuie sur les *contraintes* et les *assertions*, et gérée par les *écrivains* et *ordonnanceurs*. - La synchronisation des données est gérée par les *adaptateurs*.	- La synchronisation du temps s'appuie sur les *directeurs*. - La synchronisation des données est gérée par les *receveurs*.
La représentation du temps	Horloge globale gérée par les *quantités*.	Horloge globale gérée par les *écrivains*.	Horloge distribuée gérée par les *directeurs* instanciés à chaque niveau hiérarchique.
Avantages	Calcul et communication sont clairement séparés.	- Calcul et communication sont clairement séparés. - Introduction de la modélisation hiérarchique.	- Calcul et communication sont clairement séparés. - Modélisation entièrement hiérarchique. - Gestion hiérarchique de la synchronisation et des horloges locales à chaque domaine. - Les domaines sont clairement séparés. - Sémantique abstraite pour le contrôle de la simulation.
Inconvénients	- Absence de hiérarchie. - La synchronisation et l'horloge globale sont à la charge du concepteur au niveau langage. - Les différents domaines ne sont pas clairement séparés.	- La synchronisation et l'horloge globale sont à la charge du concepteur au niveau langage. - Les MoCs ne sont pas clairement séparés.	L'instanciation des éléments pour contrôler la synchronisation des temps à travers la hiérarchie (*directeurs*) est sous la responsabilité des concepteurs.

Table C.1: Résumé des caractéristiques des environnements de modélisation et simulation de systèmes multi-disciplines s'appuyant sur les Meta-Modèles et les Langages de programmation haut-niveau.

Environnement	HetSC	HetMoC	ForSyDe
La modélisation hiérarchique	Multi-niveaux via les composants SystemC.	Non autorisée.	Multi-niveaux via les *processus* (modules SystemC).
Séparation entre le calcul et la communication	- Le calcul s'appuie sur les *processus*. - La communication s'appuie sur les *ports*, *interfaces* et *canaux*.	- Le calcul s'appuie sur les *processus*. - La communication s'appuie sur les *signaux*.	- Le calcul s'appuie sur les *processus*. - La communication s'appuie sur les *signaux*.
Degré d'hétérogénéité	Langage.	Langage.	Langage.
La synchronisation	Utilise les *MoC interfaces*.	Utilise les *domain interfaces*.	Utilise les *domain interfaces*.
La représentation du temps	- Horloge globale du noyau de SystemC. - Restrictions définies dans les canaux propres à chaque MoC.	Information non disponible.	Encapsulée dans le constructeurs des processus propres à chaque MoC, pour un noyau de simulation unique.
Avantages	- Calcul et communication sont clairement séparés. - Modélisation hiérarchique. - Horloge globale du noyau SystemC DE. - Bibliothèque d'éléments prédéfinis pour contrôler la synchronisation. - Les interactions entre domaines sont gérées via des éléments dédiés (*MoC interfaces*).	- Calcul et communication sont clairement séparés. - Les interactions entre domaines sont gérées via des éléments dédiés (*domain interfaces*).	- Calcul et communication sont clairement séparés. - Modélisation hiérarchique. - Chaque MoC a sa propre gestion du temps, appliquée à un noyau de simulation unique. - Les interactions entre domaines sont gérées de façon propre à chaque MoC via des éléments dédiés (*domain interfaces*). - Sémantique abstraite de simulation.
Inconvénients	- L'instanciation des éléments qui contrôlent la synchronisation (*MoC interfaces*) peut être implémentée ou redéfinie par le concepteur.	- Absence de modélisation hiérarchique. - La représentation du temps n'est pas clairement définie. - L'instanciation des éléments qui contrôlent la synchronisation est à la charge du concepteur.	- Le choix et l'instanciation des éléments qui contrôlent la synchronisation (*domain interfaces*) sont à la charge du concepteur.

Table C.2: Résumé des caractéristiques des environnements de modélisation et simulation de systèmes multi-disciplines s'appuyant sur SystemC.

C.4. La synchronisation entre le domaine des événements discrets et le domaine du temps discret (chapitre 4)

C.4.1. Problème de synchronisation entre un cluster TDF et un système DE

Lorsqu'un modèle met en jeu des sous-systèmes TDF et des sous-systèmes DE, il faut garder à l'esprit que lors de la simulation, le temps DE doit être monotone croissant. Ce principe ne doit pas être modifié par des événements issus du cluster TDF. Ces événements, que l'on nomme actions de synchronisation entre DE et TDF, sont soit les opérations de lecture par un cluster TDF de données venant d'un module SystemC/DE, soit les opérations d'écritures de données depuis un cluster TDF vers un module SystemC/DE. Pour qu'un ordonnancement statique d'un cluster TDF soit valide, il doit donc respecter ce principe de monotonie, et donc des événements issus de ce cluster TDF ne peuvent se produire à un temps précédent l'horloge courante SystemC/DE.

Or, dans la PoC existante, l'ordonnancement d'un cluster TDF se fait indépendamment de l'horloge SystemC/DE. Il existe des cas où le principe de monotonie de SystemC/DE n'est pas respecté, qui entraînent une erreur lors de la simulation. Un exemple simple est présenté à la Figure C.6 (a). Le problème de causalité est le suivant. Dans le cas (b): la lecture par le module **A** du signal **sig1** à 6 ms fait avancer l'horloge de SystemC à 6 ms (⑥). La deuxième exécution du module **B** quant à elle, produit un échantillon à 4 ms (⑧) qui essaie d'écrire une donnée sur le port d'entrée du module SystemC **Y** à 4 ms également. Comme l'horloge de SystemC est déjà à 6 ms (⑤), cela crée le problème (⑨). Ce problème est résolu par l'ajout, à la Figure C.6 cas (c), d'un retard D_{out} de 1 sur le port de sortie du module **B**.

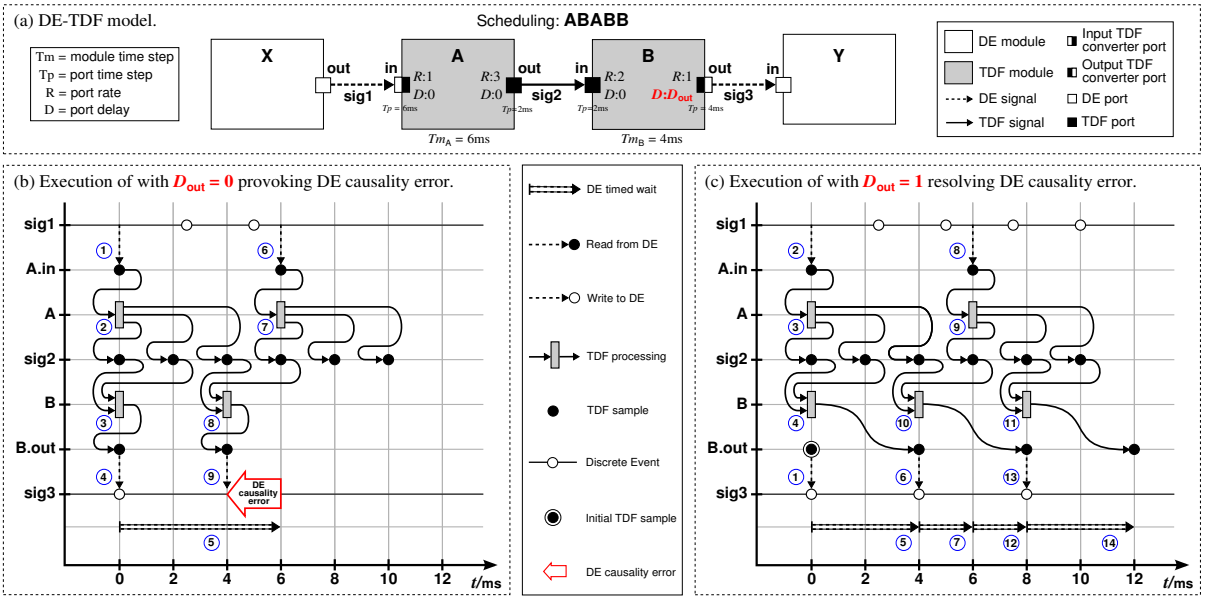

Figure C.6: Simulation d'un cluster TDF avec problème de synchronisation DE-TDF. Cas (a) le modèle DE-TDF. Cas (b) le retard $D_{out} = 0$ crée un problème de causalité. Cas (c) le retard $D_{out} = 1$ produit une simulation sans erreur.

Cette étude montre que des problèmes de causalité vont apparaître dans les modèles où les ports TDF à taux d'échantillonnage multiples, interagissent avec des modules SystemC/DE. Ces problèmes apparaissent car la synchronisation entre TDF et DE n'est pas prise en compte lors de l'ordonnancement statique des clusters TDF. La représentation graphique utilisée à la Figure C.6

permet de représenter l'évolution d'un système de petite taille, mais elle ne passe pas l'échelle d'un système plus complexe. Nous mettons ainsi en évidence la nécessité de représenter et de formaliser l'interaction TDF/DE. L'approche que nous proposons est présentée au paragraphe suivant.

C.4.2. Modélisation par un Réseau de Pétri Coloré (CPN)

Le MoC TDF s'appuie sur le modèle des SDF [35], [61] comme nous l'avons vu. Les SDF peuvent être représentés par un réseau de Petri [62]. Ces réseaux sont également bien adaptés à l'étude des MoCs TDF. Comme on le présente à la Figure C.7, les modules TDF sont modélisées par des *transitions*, les signaux par des *places*, les ports TDF par des *arcs dirigés*, les échantillons des signaux TDF comme des *jetons*, le taux d'échantillonnage d'entrée par le *nombre de jetons lus* par une transition, le taux d'échantillonnage de sortie par le *nombre de jetons écrits* par une transition, les retards par des *jetons présents initialement* (ou *marquage initial*) dans les places associées aux transitions.

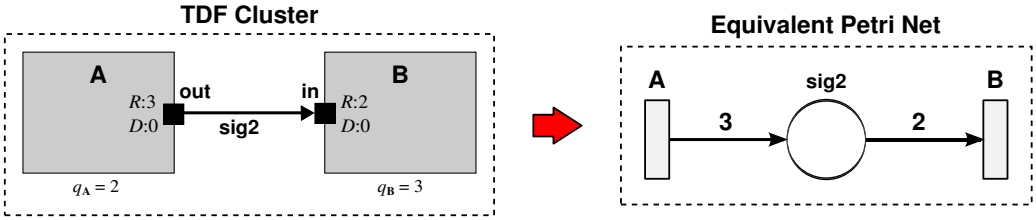

Figure C.7: Cluster TDF et son réseau de Petri équivalent.

A l'aide de ce modèle, on peut établir l'ordonnancement d'un cluster purement TDF, sans interaction DE-TDF, en suivant les règles des réseaux de Petri et du standard TDF :

- Une transition **T** d'un réseau de Petri peut être franchie si elle est validée.
- Une transition **T** d'un réseau de Petri est validée lorsque les jetons (ou marques) requis par le franchissement sont disponibles dans la place correspondante.
- Un module TDF **M** (représenté par une transition **T**) doit être exécuté un nombre de fois q_M par période d'exécution du cluster correspondant.
- Un module TDF **M** (représenté par une transition **T**) est exécuté immédiatement dès que le nombre d'échantillons sur les ports d'entrée est identique au taux d'entrée déclaré sur les ports d'entrée.

Ce modèle permet de calculer un ordonnancement statique d'un cluster exclusivement TDF, sans interaction avec DE. Le calcul de cet ordonnancement détermine également le nombre maximum d'échantillons (ou jetons ou marques) stockés dans les places durant l'exécution du cluster durant une période. Il n'utilise pas les informations temporelles, ou dates, des échantillons. Pour traiter l'interaction avec l'horloge de SystemC, il faut donc ajouter à ce modèle une information supplémentaire, qui autorise la représentation du temps. Nous avons retenu une extension de la représentation des SDF par réseau de Petri, qui est la représentation par réseaux de Petri colorés [17] et temporisés [64] présentés à la Figure C.8.

Comme l'indique la Figure C.8, un réseau de Petri coloré et temporisé, (CPN), est une représentation qui met en oeuvre des *places*, des *transitions*, des *arcs dirigés*, des *jetons colorés*, et des *annotations*.

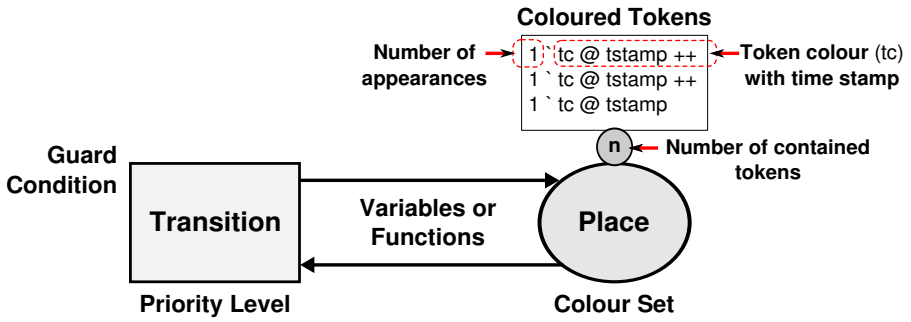

Figure C.8: Exemple d'un réseau de Petri coloré.

- **Les transitions** : ce sont les actions effectuées par les modules. Elles ont un degré de priorité (*priority levels*) et leur franchissement peut être soumis à une condition (*guard condition*).
- **Les places** : elles représentent l'état d'un modèle. Cet état est caractérisé par le nombre de jetons contenus dans une place, ainsi que par le type de donnée associée au jeton, déterminée par sa couleur (*token colour*). Dans un réseau CPN, l'état d'un modèle évolue suivant le franchissement des transitions.
- **Les arcs dirigés** : ils relient une place à une transition, soit une transition à une place. Une variable ou une fonction peut être associée à un arc pour préciser l'évolution de l'état du modèle après franchissement de la transition.
- **Les jetons (ou marques) colorés** : ils représentent les échantillons multiples. Au jeton est associé un type de donnée, appelée "la couleur". On la note par un type (i.e. entier, nom, booléen) à droite de l'opérateur " ` " , le nombre d'occurrence étant, lui, noté à gauche de cet opérateur. A un jeton peut être associée une deuxième information, appelée date, qui indique à quel instant le jeton est susceptible d'être consommé par la transition.
- **Les annotations** : elles sont exprimées dans le langage de programmation CPN ML [63] et peuvent être associées à une transition, une place ou un arc. Elles permettent de caractériser :
  - **une transition** : par son degré de priorité (*priority level*) ou une condition de franchissement (*guard condition*).
  - **une place** : par le type de données contenues dans la place (*colour set*).
  - **un arc** : en exprimant une fonction évaluée au cours de la simulation, qui peut consommer et produire des jetons.

Pour traiter le problème de synchronisation entre un modèle SystemC/DE et un modèle TDF, nous avons proposé une représentation équivalente du système DE/TDF sous la forme d'un réseau de Petri coloré temporisé (CPN). Nous présentons à la Figure C.9 les primitives de cette représentation.

- **Le modèle équivalent CPN d'un module TDF**. Un module type est représenté à la Figure C.9, partie (a). Le nom **M:**q_M, identifie la transition (i.e. le module TDF). La condition de franchissement $[j_M <> q_M]$, est un booléen fonction de j_M, le nombre de fois où le module **M** a été exécuté, et q_M le nombre de fois où le module **M** doit être exécuté pendant une période TDF. Quand la transition est franchie, l'index j_M est incrémenté.

- **Le modèle équivalent CPN des connexions entre modules TDF**. Une connexion type est représentée à la Figure C.9, partie (b). $\mathbf{S_N}$ est le nom du signal qui relie le module source **M** à l'entrée du module destination **N**. **m** est le nom du port de sortie appartenant au module **M**, **n** est le nom du port d'entrée appartenant au module **N**. **R** est le taux d'échantillonnage multiple défini pour un port, **D** est le retard défini pour un port, en nombre d'échantillons. Le marquage initial $D_{\mathsf{m+n}}$ de la place $\mathbf{S_N}$ s'exprime en fonction de D_{m} et D_{n}. Après franchissement d'une transition $\mathbf{M}{:}\boldsymbol{q}_{\mathbf{M}}$ ou $\mathbf{N}{:}\boldsymbol{q}_{\mathbf{N}}$, des fonctions d'incréments mettent à jour les index respectifs j_{M}et j_{N} en fonction des paramètres **R** et **D** des ports correspondants.

- **Le modèle équivalent CPN d'un port de conversion à l'entrée d'un module TDF**. Un port de conversion d'entrée type est représenté à la Figure C.9, partie (c). Comme ce composant synchronise DE et TDF, son modèle comporte des informations temporelles explicites. Il s'agit du pas de temps Tm_{M} du module **M** et du pas de temps Tp_{m} du port **m**. Plusieurs fonctions utilisent ces valeurs, pour construire le modèle équivalent CPN temporisé, dans lequel :

  - Les places **S read ops. list** et **S read op. enabled** enregistrent respectivement les événements de synchronisation de lecture d'un signal et les lectures effectivement autorisées au temps courant t_{CPN}. La place **M.m** enregistre les jetons (échantillons) qui peuvent être consommés par la transition $\mathbf{M}{:}\boldsymbol{q}_{\mathbf{M}}$. Cette place est la base du port de conversion.
  - Les transitions **Enable S read op.** et **Read S** effectuent respectivement la validation de la lecture au temps t_{CPN} et la lecture effective d'un signal depuis DE vers TDF.
  - Il convient de noter l'apparition d'arcs inhibiteurs entre une transition **T** et une place **P**. Ils explicitent un franchissement conditionnel : la transition **T** ne peut être franchie qu'à la condition que la place **P** soit vide. Ici chaque transition de type **Enable S read op.** est conditionnée par les places **S read op. enabled** et **S write op. enabled**.

- **Le modèle équivalent CPN d'un port de conversion à la sortie d'un module TDF**. Un port de conversion de sortie type est représenté à la Figure C.9, partie (d). Il comporte également des informations temporelles nécessaires à la synchronisation TDF/DE.

  - Les places **S write ops. list** et **S write op. enabled** enregistrent respectivement les événements de synchronisation d'écriture d'un signal et les écritures effectivement autorisées au temps courant t_{CPN}. La place **M.m** enregistre les jetons qui doivent être écrits dans le domaine DE, depuis le domaine TDF par la transition $\mathbf{M}{:}\boldsymbol{q}_{\mathbf{M}}$. Cette place est la base du port de conversion **m** du module **M**. Il convient de noter la présence d'un retard D_{m} comme attribut de ce port qui sera utilisé pour réaliser l'ordonnancement d'un cluster TDF dans un système DE.
  - Les transitions **Enable S write op.** et **Write S** effectuent respectivement la validation de l'écriture au temps t_{CPN} et l'écriture effective d'un signal depuis TDF vers DE.
  - Les arcs inhibiteurs des places **S write op. enabled** conditionnent le franchissement de la transition **Enable S write op.**

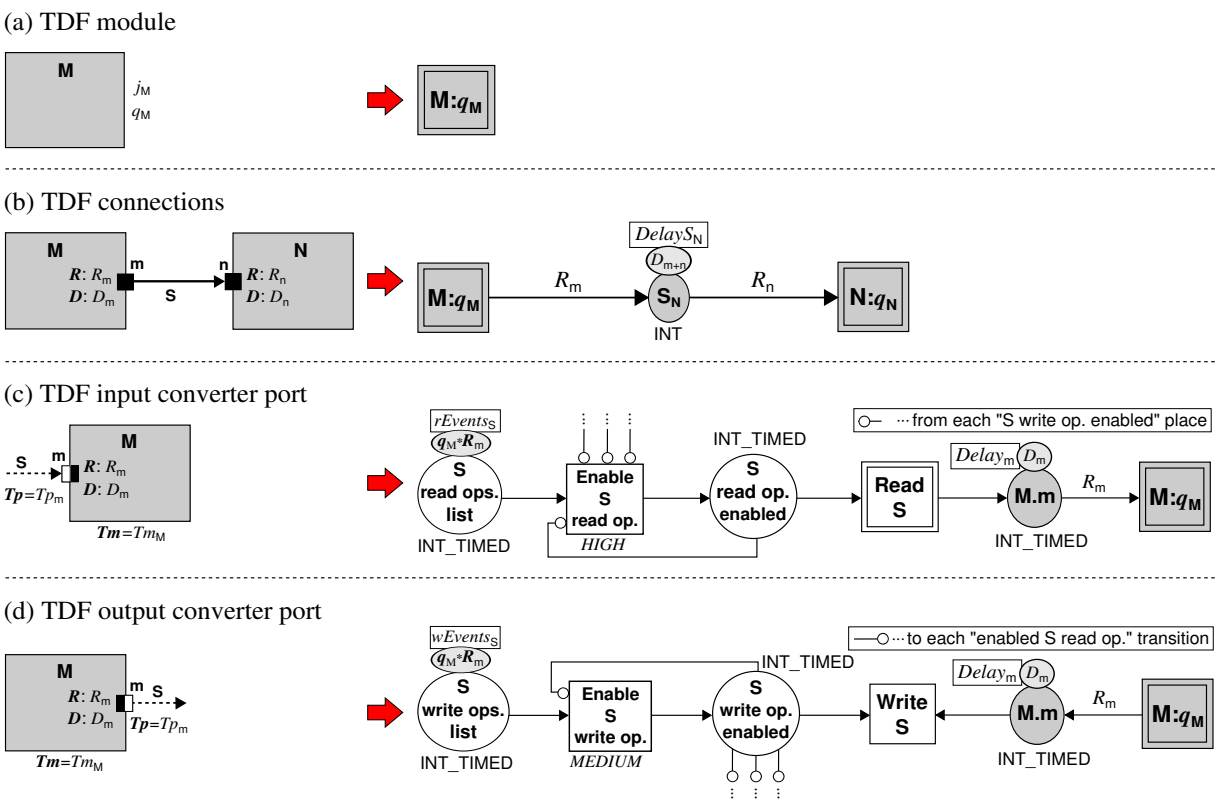

Figure C.9: Cluster TDF et son réseau de Petri coloré temporisé équivalent CPN.

C.4.3. Exemple d'analyse DE-TDF avant la simulation

A l'aide des principes énoncés au chapitre 4, il est possible d'établir le réseau de Petri coloré temporisé CPN de l'exemple présenté à la Figure C.6. Ce modèle équivalent CPN est présenté à la Figure C.10. Il a été validé par l'outil d'édition de CPN [65]. L'exécution de ce modèle avec $D_{\text{out}} = 1$ conduit au chronogramme de la Figure C.6(c) et son exécution avec $D_{\text{out}} = 0$ conduit au chronogramme Figure C.6(b), interrompu du fait d'une erreur de synchronisation DE-TDF.

En nous appuyant sur les règles d'exécution des réseaux de Petri CPN et des restrictions imposées par le standard TDF, nous avons développé une méthode d'analyse des modèles équivalents CPN, puis détecté les erreurs de synchronisation éventuelles et finalement proposé une correction. Cette méthode est présentée dans le paragraphe suivant et dans la version complète anglaise à la Section 4.4.

C.4.4. Modélisation du problème de synchronisation par un CPN

Nous montrons ici comment il est possible d'utiliser le modèle équivalent CPN d'un système DE-TDF pour vérifier qu'il existe un ordonnancement statique de ce système. Suivant les principes énoncés précédemment, vérifier qu'il existe un ordonnancement statique du système DE-TDF consiste à vérifier que le réseau équivalent CPN est franchissable sans interruption pendant une période `T_cpn` du cluster TDF. Cette analyse est effectuée en 3 phases :

- **Phase 1 – Franchissement des transitions**. Pendant cette phase, toutes les transitions qui peuvent l'être sont franchies.

– Les transitions sont franchies suivant leur degré de priorité.

– En s'appuyant sur les transitions de degré de priorité le plus bas, qui représentent les transitions des modules TDF (**M:q_M**) et les interactions entre modules TDF et modules DE (transitions **Read S** et **Write S**), on construit un ordonnancement statique du système.

– On augmente le temps t_cpn d'exécution du réseau CPN, sans dépasser la période T_cpn.

- **Phase 2 – Etat final atteint** . On vérifie si l'état final est atteint. 2 cas peuvent se présenter :
  – *L'état final est atteint à la première exécution*: il existe un ordonnancement du système. L'analyse est terminée.
  – *L'état final n'est pas atteint.* Il existe un problème de causalité. L'analyse passe en phase 3.
  – *L'état final est atteint, après détection d'une erreur de synchronisation.* Les problèmes de synchronisation ont été détectés et corrigés à la phase 3. Un diagnostic est adressé au concepteur et l'analyse est terminée.

- **Phase 3 – Conditions de blocage et correction**. On identifie les composants du CPN (transitions et places) qui sont à l'origine du blocage au temps t_cpn. On modifie les attributs de ces objets en conséquence. On relance la phase 1 de l'analyse.

Cette analyse a été appliquée à l'exemple de la Figure C.6. Lorsque $D_{out} = 0$, l'analyse conduit à la Figure C.11. Il apparaît un problème de causalité au temps $t_{CPN} = 4$ ms dans le rectangle jaune car :

- Le réseau CPN est bloqué alors qu'il n'a pas atteint l'état final.
- La place **sig3 write op. enabled** indique d'une opération de synchronisation devrait être franchie au temps $t_{CPN} = 4$ ms.
- La place **B.out** ne contient pas de jeton disponible au temps $t_{CPN} = 4$ ms.
- La transition **Write sig3** est bloquée, et donc l'opération de synchronisation ne peut être exécutée.

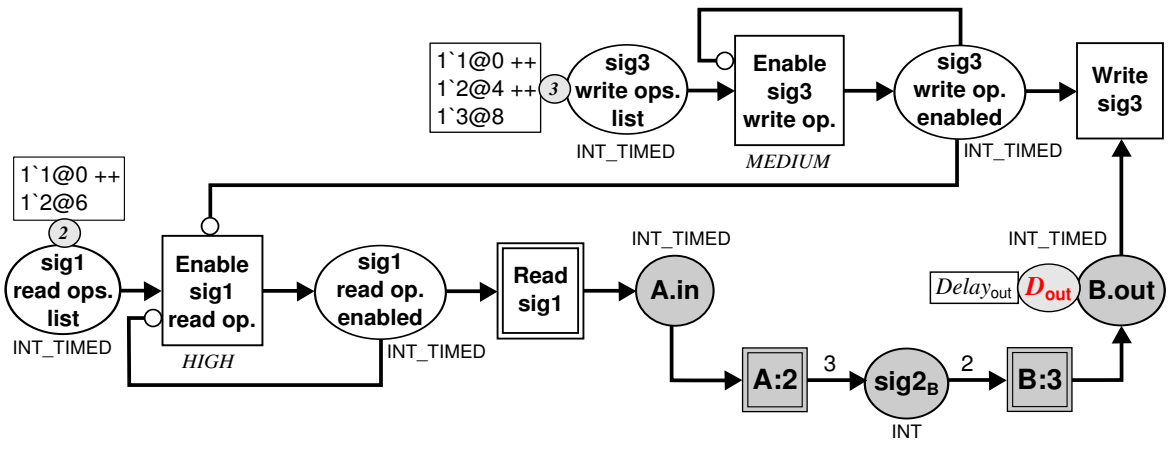

Figure C.10: Modèle équivalent CPN du modèle DE-TDF de la Figure C.6(a).

Une fois l'erreur détectée, elle est corrigée en repérant la transition fautive **Write S**, puis en supprimant les jetons de la place correspondante **S write op. enabled** et en incrémentant le retard de la place correspondante **M.m** du nombre de jetons présents dans la place **S write op. enabled**. Ce qui se traduit dans l'exemple considéré par :

- Le jeton "2@4" est supprimé de la place **sig3 write op. enabled**.
- Le retard de la place **B.out** est incrémenté de 1 ($D_{\text{out}} = 0 \rightarrow D_{\text{out}} = 1$).

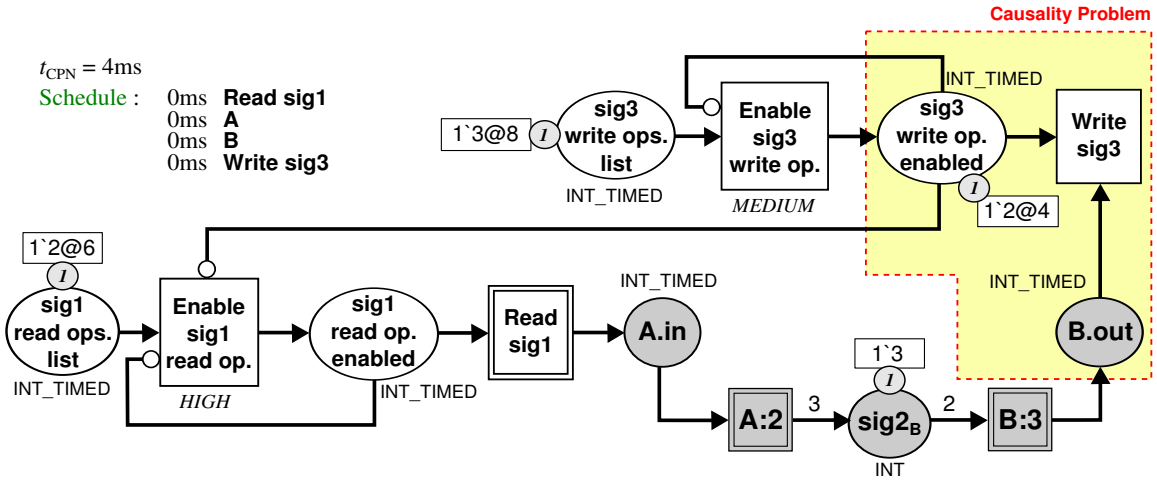

Figure C.11: Détection des problèmes de synchronisation du modèle équivalent CPN présenté à la Figure C.10 dans le cas $D_{\text{out}} = 0$.

Si cette approche s'est révélée très efficace pour traiter la synchronisation entre les domaines DE et TDF, elle ne permet pas d'appréhender la synchronisation de DE avec d'autres domaines temporels. C'est pourquoi nous présentons au paragraphe suivant et au chapitre 5 de la version anglaise complète, un environnement de simulation plus général, destiné à la simulation de systèmes multi-disciplines et multi-domaines temporels.

C.5. Le prototype du simulateur SystemC MDVP (Multi-Disciplinary Virtual Prototyping) (chapitre 5)

Ce chapitre présente les principes de construction d'un environnement de simulation multi-disciplines, s'appuyant sur SystemC pour la simulation du domaine DE. Les principes qui sont énoncés ici ont été largement discutés au sein du LIP6 dans le cadre du projet européen Catrene H-Inception [66].

C.5.1. Définition d'un modèle de calcul (MoC) de SystemC MDVP

Comme nous l'avons vu au chapitre C.3, plusieurs auteurs ont abordé ce problème, il est donc nécessaire de définir précisément les notions que nous allons utiliser dans la suite de ce document.

Nous introduisons la notion de **Modèle de Calcul (MoC)** pour définir la sémantique de modélisation d'un système, qui concerne *l'abstraction du temps, l'élaboration d'un modèle de SoC, les communications entre sous-systèmes du SoC , la synchronisation entre domaines temporels distincts, les phases d'élaboration et de simulation* au sens de SystemC.

Plus précisément :

- **L'abstraction du temps** définit la représentation du temps valide pour modéliser les composants d'un MoC (i.e. temps continu, temps discret, temps échantillonné).
- **La sémantique d'élaboration d'un modèle de SoC** définit comment manipuler les différents composants d'un SoC pour créer un modèle simulable. En SystemC MDVP, ce traitement repose sur les *modules*.
- **La sémantique de communication** définit les interactions entre composants d'un même MoC. En SystemC MDVP, ces interactions sont construites avec des *ports*, des *interfaces* et des *canaux*.
- **La sémantique de synchronisation** définit les interactions entre composants représentés suivant différents MoCs, utilisant des domaines temporels distincts. En SystemC MDVP, ces interactions sont déterminées par les *solveurs*.
- **La sémantique d'élaboration et de simulation** définit les étapes d'analyse préalables à l'exécution effective de la simulation. En SystemC MDVP, ces traitements sont également à la charge des *solveurs*.

En SystemC MDVP, les *modules*, *ports*, *interfaces* et *canaux* sont instanciés par le concepteur pour écrire le comportement d'un SoC; mais les *solveurs* sont instanciés automatiquement par le simulateur.

C.5.2. Principe de modélisation en SystemC MDVP

Le modèle SystemC MDVP d'un SoC multi-disciplines est construit en interconnectant des *modules*, appartenant à différents MoCs, à l'aide de *ports*, *interfaces* et *canaux* relatifs à un certain MoC. La Figure C.12 illustre la vision d'un concepteur de modèle de SoC dans un cas typique.

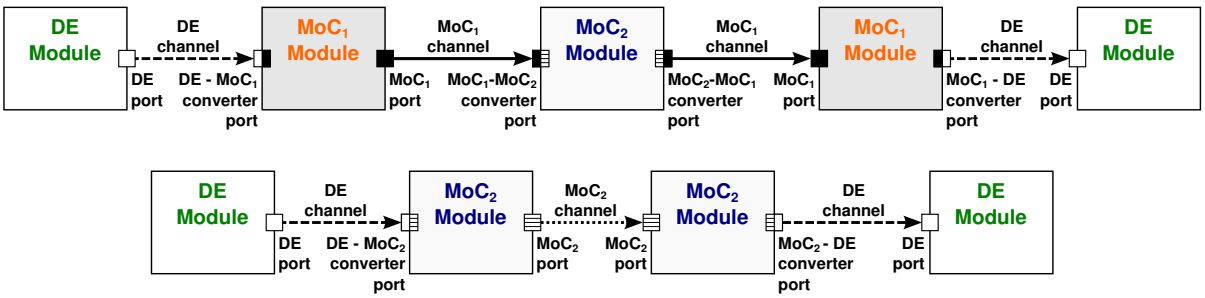

Figure C.12: Composants SystemC MDVP.

Précisons encore ces termes :

- **Les modules** : sont les objets qui traitent le comportement suivant les règles et algorithmes propres à un certain MoC. Ils possèdent des ports d'entrée et sortie par lesquels ils communiquent des données.
- **Les ports** : sont les objets utilisés par les modules pour échanger des données avec d'autres modules qu'ils soient, ou non, relatifs au même MoC. Les ports assurent la communication et la synchronisation des données échangées. Ainsi on a défini :
  - **Ports classiques** : sont les objets utilisés par deux modules du même MoC pour échanger des données.

  - **Ports de conversion** : sont les objets utilisés par deux modules de deux MoCs distincts pour échanger des données. On a ainsi :
    * **Ports de conversion d'entrée** : d'un module du $\mathbf{MoC_2}$ qui reçoit des données d'un module du $\mathbf{MoC_1}$ par un canal de type $\mathbf{MoC_1}$.
    * **Ports de conversion de sortie** : d'un module du $\mathbf{MoC_2}$ qui veut transmettre des données à un module du $\mathbf{MoC_1}$ par un canal de type $\mathbf{MoC_1}$.

- **Interfaces et Canaux**. Les *interfaces* sont les fonctions utilisées pour accéder aux *canaux* qui contiennent les données échangées par les modules. Un canal est relatif à un certain MoC, cependant, via les ports de conversion, il peut être utilisé pour interconnecter des modules de différents MoCs.

Soulignons que cette représentation assure une séparation claire entre le calcul du comportement, inclus dans le module et les communications gérées par les ports, les interfaces et canaux.

Soulignons également que cette approche autorise une modélisation hiérarchique. On appelle cluster, un ensemble de modules interconnectés relatifs à un même MoC. La Figure C.13 illustre la hiérarchie dont la construction suit les règles suivantes :

- La hiérarchie d'un modèle SystemC MDVP suit la hiérarchie des MoCs instanciés.
- Les clusters sont considérés comme des boîtes noires qui se comportent comme les modules du niveau hiérarchique (MoC) avec lesquels ils sont interconnectés.
- Les clusters peuvent contenir des modules ou des clusters.
- Les limites d'un cluster sont définies par les ports de conversion qui synchronisent deux MoCs : celui qui instancie le cluster et celui du cluster lui même.

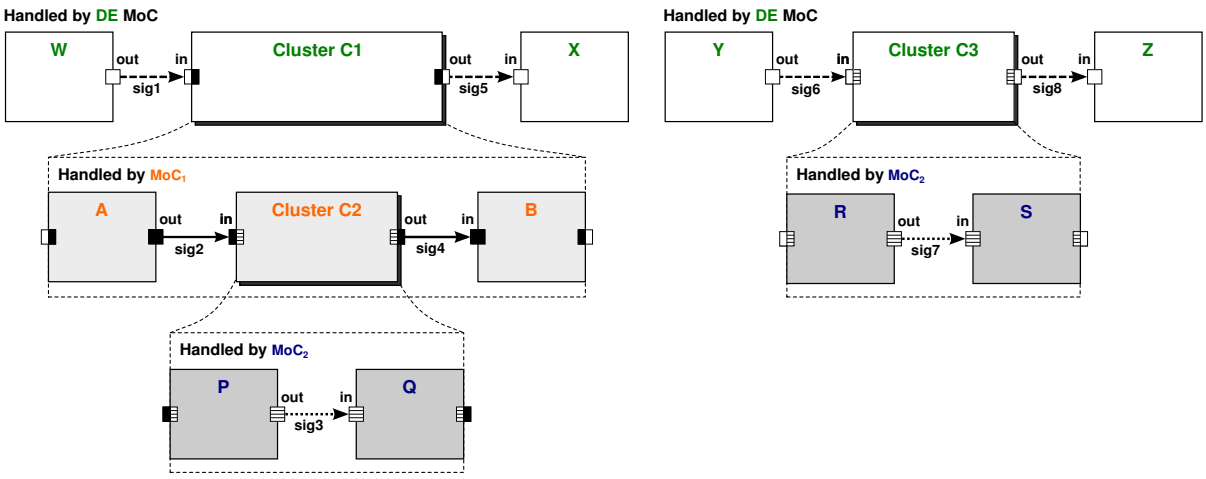

Figure C.13: Encapsulation de modules SystemC MDVP dans des clusters, pour l'exemple de la Figure C.12.

C.5.3. Définition d'un solveur en SystemC MDVP

Un **solveur** de SystemC MDVP est défini par le concepteur du MoC. Son rôle est de :

- traiter la synchronisation temporelle entre une paire de 2 MoCs, l'un étant le *maître* et l'autre *l'esclave*. Le MoC *maître* impose les contraintes de synchronisation et *l'esclave* est le MoC qui contient le solveur.
- traiter les phases d'élaboration et de simulation des composants qui relèvent de ce MoC, conformément au noyau de SystemC.

Alors que la POC SystemC-AMS existante repose sur une couche de synchronisation unique entre DE et TDF, nous proposons ici une approche générique de la synchronisation des MoCs qui est illustrée à la Figure C.14. On souligne le fait que le **DE MoC** (1 à la Figure C.14) du standard SystemC est considéré dans cette étude comme la base pour établir les mécanismes de synchronisation avec d'autres MoCs.

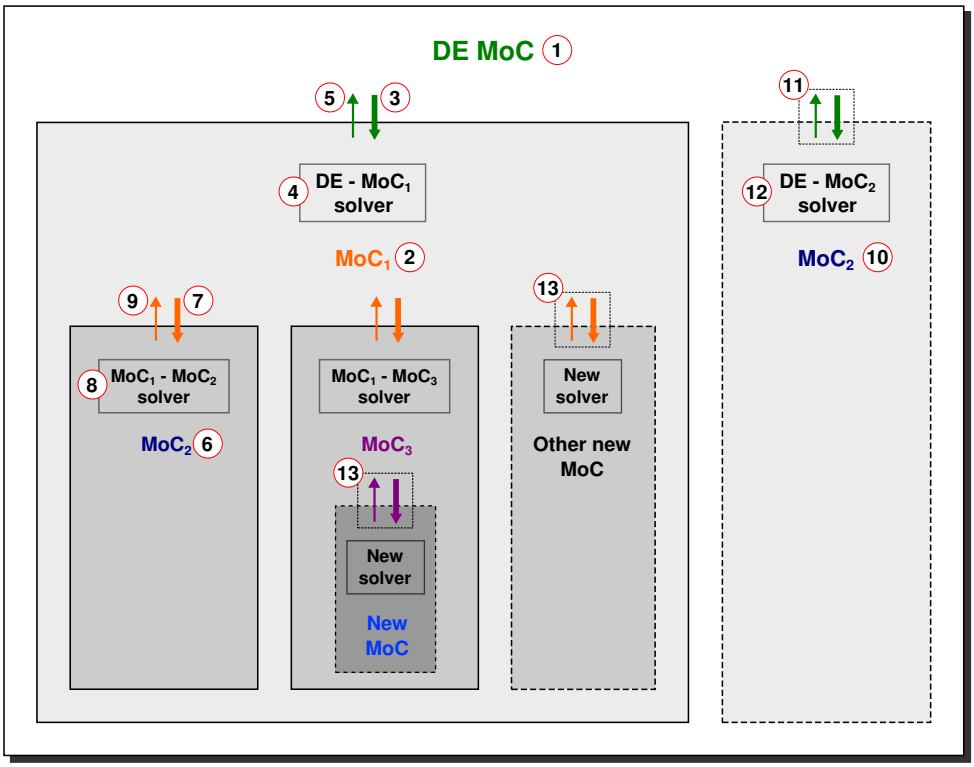

Figure C.14: Proposition d'architecture pour le simulateur SystemC MDVP.

Considérons par exemple les interactions entre le **DE MoC** et le **MoC$_1$** (2 à la Figure C.14). Elles sont traitées par le **solveur DE-MoC$_1$** (4 à la Figure C.14). Ainsi :

- Pendant la phase d'*élaboration*, les instances du **solveur DE-MoC$_1$** effectuent l'analyse et la préparation des clusters qui relèvent du **MoC$_1$** qui doivent interagir avec **DE MoC**. Les solveurs qui doivent traiter les interactions entre plusieurs MoCs sont enregistrés dans un dictionnaire. Ainsi la Figure C.14 aura, une fois la phase d'élaboration complétée, et les clusters créés, pour dictionnaire de solveurs :

Paire de MoCs < master, slave >	Solveurs
< DE, MoC_1 >	solveur DE - MoC_1
< DE, MoC_2 >	solveur DE - MoC_2
< MoC_1, MoC_2 >	solveur MoC_1 - MoC_2

Table C.3: Dictionnaire de solveurs pour l'exemple de la Figure C.14.

- Pendant la *simulation*, les instances du **solveur DE-MoC_1** traitent la synchronisation temporelle en 3 phases :
  - Premièrement, le noyau DE impose les *contraintes de synchronisation temporelles* pour l'exécution du cluster **MoC_1** (③ à la Figure C.14).
  - Deuxièmement, les instances du **solveur DE-MoC_1** traitent l'exécution des composants du cluster **MoC_1** (④ à la Figure C.14).
  - Troisièmement, quand les instances du **solveur DE-MoC_1** rencontrent une *action de synchronisation*, elles rendent le contrôle au noyau de simulation DE (⑤ à la Figure C.14), via une instruction wait() pour être réactivées dans le futur.

Pour assurer la cohérence des règles de simulation entre les modules et les solveurs d'un même MoC, on a recours à une sémantique abstraite. Ainsi :

- Chaque module doit implémenter la *sémantique* définie par le MoC dont il relève.
- Chaque solveur d'un MoC_2 doit implémenter la *sémantique* définie par le MoC_1 avec lequel les composants du MoC_2 veulent communiquer.

Dans le cas des modules DE, l'élaboration et la simulation sont définies par le noyau de SystemC. Nous avons donc défini une classe appelée **DE MoC interface** pour traiter les solveurs qui veulent échanger des données avec le solveur DE. La classe *DE MoC interface* est définie au moyen de 2 fonctions elaborate() et simulate(). Ces fonctions seront implémentées différemment selon la sémantique du MoC qui devra interagir avec DE. Un exemple est présenté à la Figure C.15.

C.5.4. La simulation d'un modèle SystemC MDVP

Pour lancer l'exécution de la simulation d'un modèle SystemC MDVP, le concepteur du SoC utilise, comme en SystemC, la fonction sc_start(), alors le noyau de simulation SystemC lance la phase d'*élaboration*. Pour réaliser le simulateur SystemC MDVP, nous avons étendu les phases d'*élaboration* et de *simulation* conformément au standard SystemC, comme cela est présenté par la Figure C.16. Nous avons ajouté les fonctionnalités suivantes à la fonction de rappel end_of_elaboration() :

- La création des clusters (grâce aux ports de conversion)
- L'instanciation des solveurs (suivant la hiérarchie des clusters et les paires de MoCs maître-esclave)
- L'élaboration hiérarchique de modules (attributs et ordonnancement selon les MoCs)
- L'élaboration hiérarchique des ports et des canaux (taille)

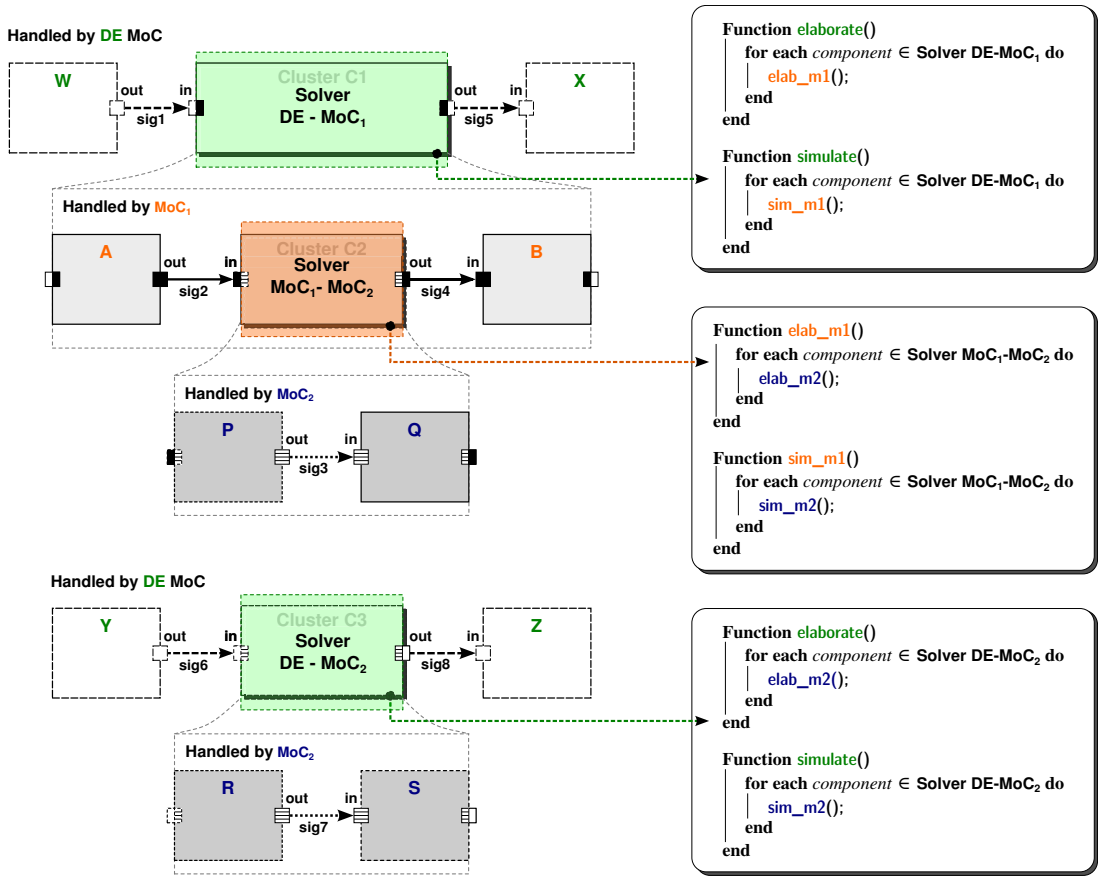

Figure C.15: Exemple d'utilisation d'une sémantique abstraite pour l'élaboration et la simulation selon SystemC MDVP.

Quant à la fonction de rappel de SystemC start_of_simulation(), nous l'avons complétée en introduisant l'initialisation des modules et des solveurs, ainsi que l'enregistrement des solveurs. Cet enregistrement crée un processus dynamique *SystemC dynamic process*, via la fonction sc_spawn().

C.5.5. Les classes de base du simulateur SystemC MDVP

Nous donnons dans ce paragraphe un bref aperçu de l'implémentation du simulateur SystemC MDVP. Cette implémentation est discutée plus en détail au chapitre 5 de la version anglaise. Nous soulignons ici quatre caractéristiques de cette implémentation :

- Pour bénéficier des fonctionnalités offertes par le noyau de simulation de SystemC, nous avons implémenté les modules, solveurs, canaux et ports comme des classes dérivées de la classe objet de SystemC **sc_object**.
- Pour traiter les phases d'élaboration et simulation par des fonctions génériques et récursives, nous avons défini une classe unique *MoC Interface* dont héritent les modules et les solveurs **sca_moc_if**.
- Le parcours de la hiérarchie des modules et des solveurs utilise les ports.
- Le lien entre les noyaux SystemC et SystemC MDVP se fait par la classe appelée Simulation Context (**sca_simcontext**).

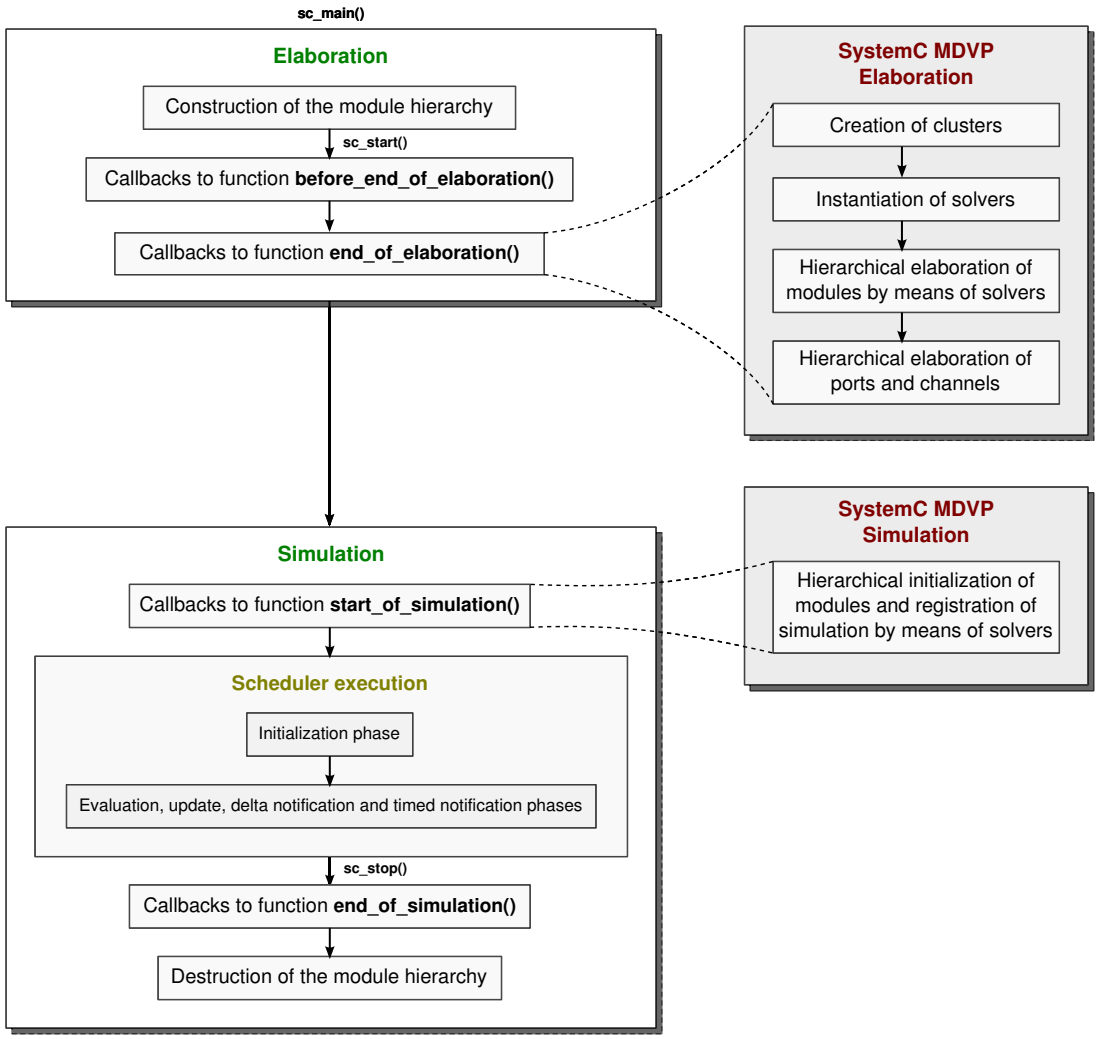

Figure C.16: Les phases d'élaboration et de simulation du simulateur SystemC MDVP fondé sur SystemC.

C.5.6. Ajout d'un modèle de calcul (MoC) au simulateur SystemC MDVP

Notre but est de définir une procédure systématique pour intégrer un nouveau MoC à SystemC MDVP et de faciliter cet ajout. Nous précisons dans ce paragraphe quelles sont les actions à mener pour l'ajout d'un MoC à SystemC MDVP, au niveau du simulateur.

- **Interface de programmation de ce MoC** (**sca_moc_if**) : il s'agit de la spécification de l'ensemble des méthodes pour réaliser les phases d'élaboration et de simulation :
  - des modules spécifiques à ce MoC,
  - des solveurs qui vont produire et échanger des données.
- **Les composants du MoC** : il s'agit de la spécification des classes des modules **sca_module**, ports **sca_port** et canaux **sca_channel**, qui héritent des classes du noyau SystemC MDVP, présentés à la Figure C.17.
- **L'emplacement du nouveau MoC dans la hiérarchie SystemC MDVP**. Il s'agit de déterminer de quel maître déjà existant dans SystemC MDVP, ce nouveau MoC va être l'esclave. Il est possible d'établir plusieurs paires de ce type. Une fois le maître choisi en fonction des interactions futures de ce nouveau MoC, il convient de déterminer :

– Les *ports de conversion* pour déterminer la *synchronisation des données* entre ce nouveau MoC et son maître.
– Un *solveur* capable de traiter les phases d'élaboration et de simulation des composants de ce MoC, et la *synchronisation temporelle* entre les données produites ou lues par ce MoC avec son MoC maître.

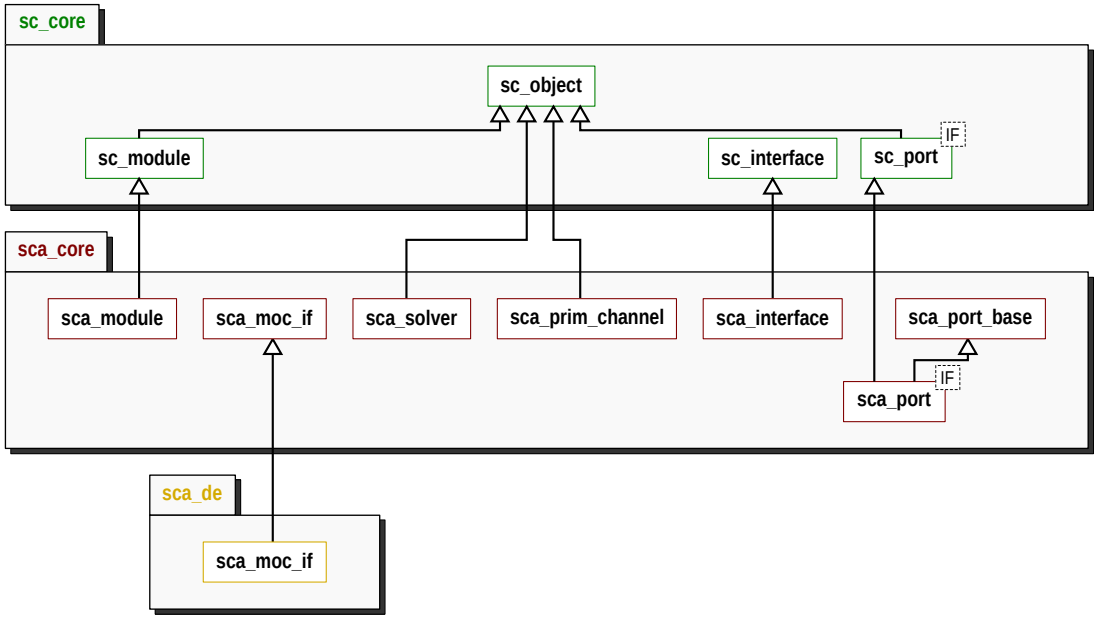

Figure C.17: Vue d'ensemble des classes du simulateur SystemC MDVP.

Note: pour conserver la comptabilité avec SystemC-AMS, nous utilisons le préfixe sca pour nommer les classes SystemC MDVP.

C.5.6.1 Modules

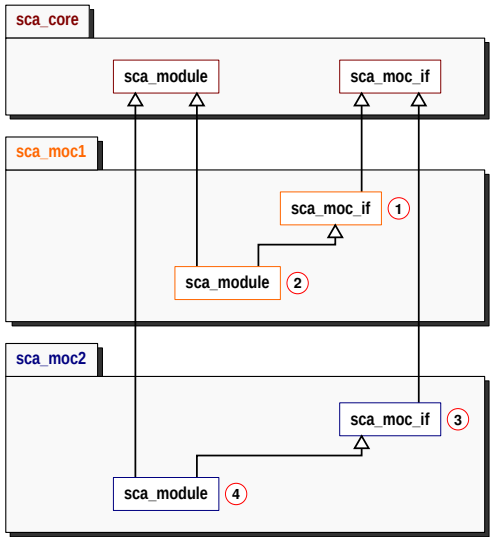

Figure C.18: Vue d'ensemble des classes pour ajouter les modules d'un nouveau MoC en SystemC MDVP.

Les Figures C.18, C.19, C.20 et C.21 présentent respectivement l'héritage des classes des modules, canaux, solveurs et les ports (classiques et de conversion) mis en œuvre dans l'implémentation de SystemC MDVP.

C.5.6.2 Canaux

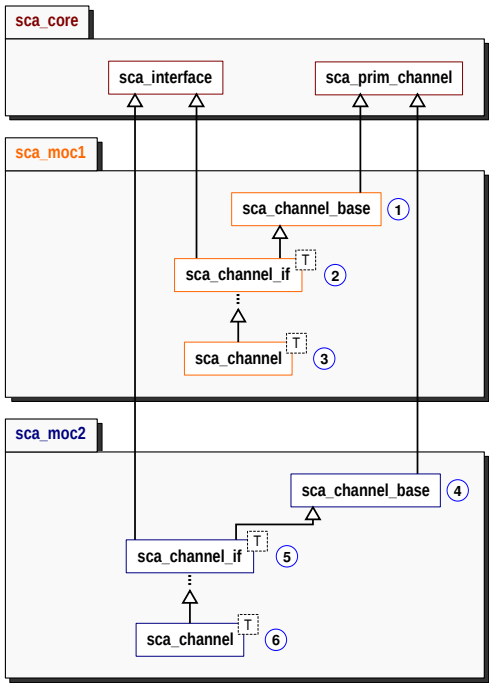

Figure C.19: Vue d'ensemble des classes pour ajouter les canaux d'un nouveau MoC en SystemC MDVP.

C.5.6.3 Solveurs

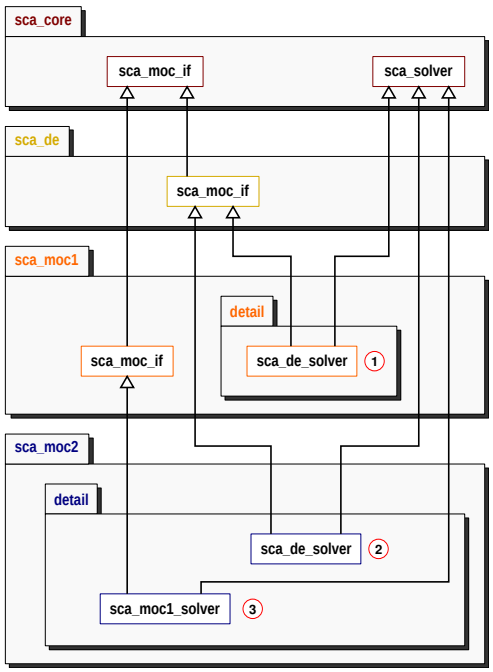

Figure C.20: Vue d'ensemble des classes pour ajouter les solveurs d'un nouveau MoC en SystemC MDVP.

C.5.6.4 Ports

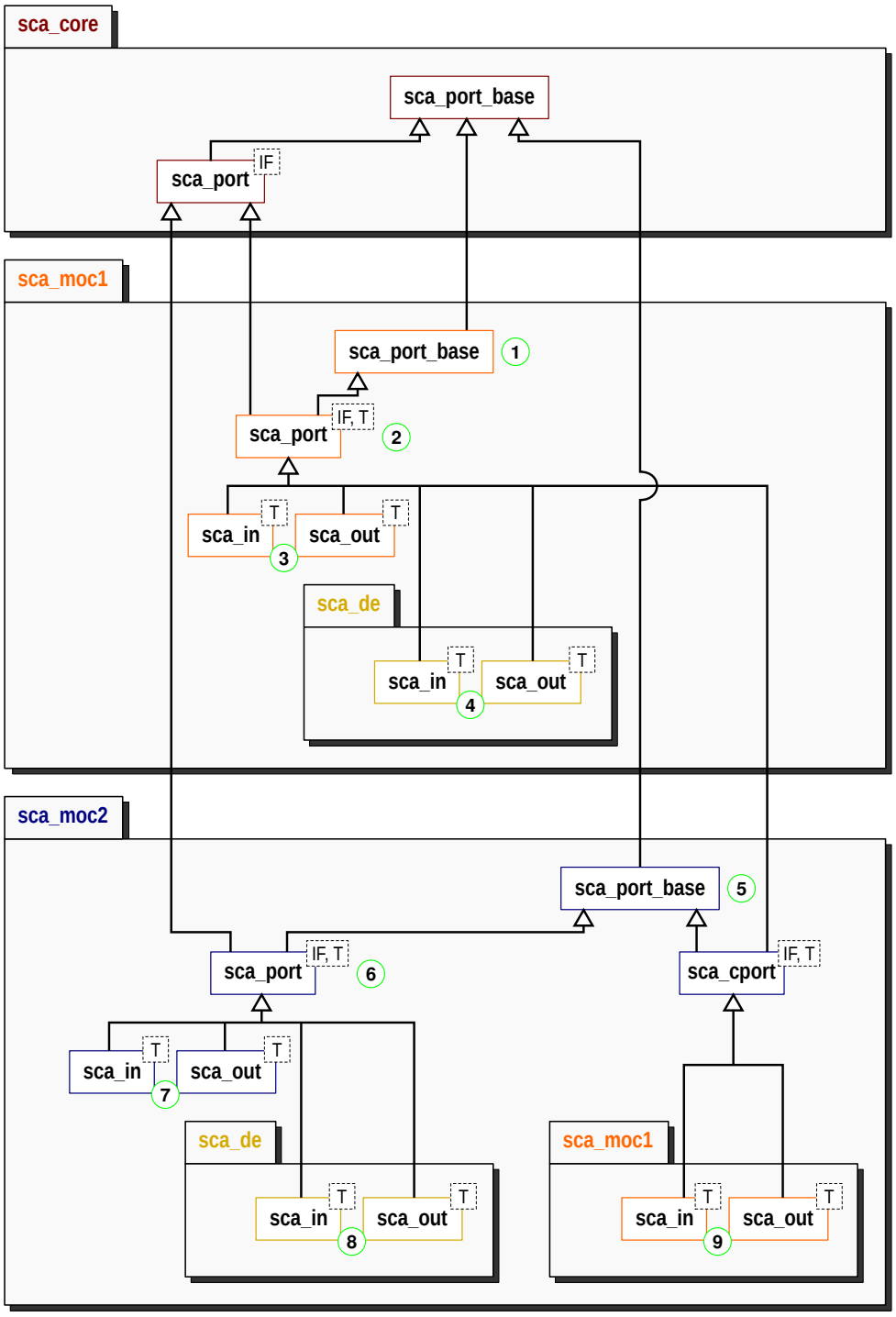

Figure C.21: Vue d'ensemble des classes pour ajouter les ports d'un nouveau MoC en SystemC MDVP.

Nous présentons ainsi notre vision d'un environnement de simulation multi-discipline. Il s'appuie sur le standard SystemC sans modifier son noyau de simulation. Nous avons vu comment modéliser un système et faire l'implémentation logicielle du simulateur. Puis nous avons proposé une approche pour rendre ce simulateur extensible, prêt à accueillir un nouveau MoC. Nous allons exploiter ces propriétés dans le paragraphe suivant en montrant comment intégrer le MoC TDF à SystemC en suivant cette approche et la modélisation par CPN équivalent du paragraphe précédent.

C.6. Le modèle de calcul TDF (Timed Data Flow) de SystemC MDVP (chapitre 6)

Dans ce paragraphe, nous allons présenter la nouvelle implémentation du MoC TDF au sein du simulateur SystemC MDVP en suivant la modélisation par réseau de Petri coloré temporisé CPN et la définition des classes d'un MoC présentées précédemment.

La première étape consiste à spécifier les *interfaces du MoC TDF* pour les phases d'élaboration et de simulation. Conformément au standard SystemC AMS, le concepteur du MoC TDF doit fournir 3 fonctions set_attributes(), pour fixer les attributs des ports et des modules utilisés dans la phase d'élaboration, initialize(), pour fixer les valeurs initiales des échantillons lors de la phase de simulation et processing() pour décrire le comportement d'un module exécuté par la phase de simulation.

C.6.1. Composants du MoC TDF de SystemC MDVP

La deuxième étape consiste à développer les composants offerts au concepteur du modèle du SoC qui relèvent du MoC TDF. Il s'agit des modules, canaux et des ports classiques TDF. La Figure C.22 présente en particulier les interactions entre ces 3 composants.

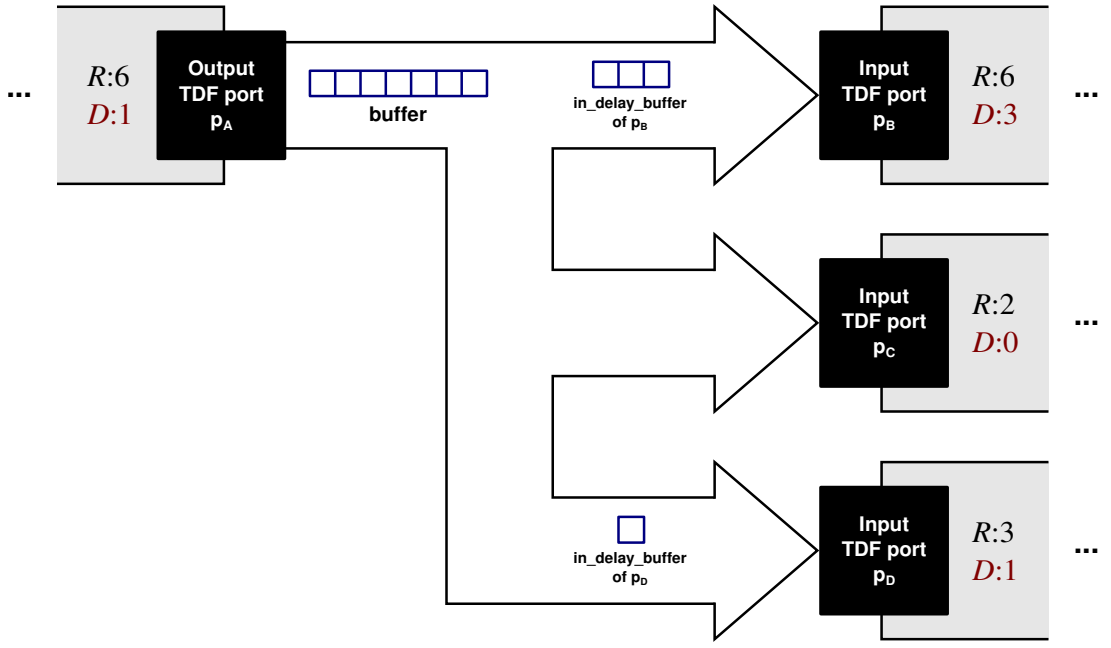

Figure C.22: Spécification d'un canal du MoC TDF de SystemC MDVP pour relier les modules TDF via les ports.

Un canal TDF peut être relié à plusieurs ports TDF si les conditions suivantes sont satisfaites :

- Un canal TDF relie au moins un port de sortie TDF à un port d'entrée TDF.
- Un canal TDF est relié à un et un seul port de sortie TDF.
- Un canal TDF est relié à un ou plusieurs ports d'entrée TDF.

Un canal contient un buffer (mémoire tampon) pour enregistrer les données écrites par le port de sortie auquel il est relié, à l'initialisation le cas échéant et au cours de la simulation. Il est instancié durant la phase d'élaboration et sa taille est déterminée en fonction du taux de sur-échantillonnage R_p du port p.

Un canal peut contenir d'autres buffers dont la taille est déterminée en fonction des retards associés à chacun des ports d'entrée auxquels le canal est relié.

C.6.2. Ports de conversion du MoC TDF de SystemC MDVP

Puisque le MoC TDF est le premier modèle de calcul qui va être ajouté à SystemC MDVP, il apparaît au deuxième niveau hiérarchique que l'on trouve à la Figure C.14. Le MoC TDF doit donc offrir en particulier :

- Des ports de conversion pour traiter la *synchronisation des données* entre le MoC TDF et le MoC DE.
- Un solveur pour traiter la *synchronisation temporelle* entre le MoC TDF et le MoC DE.

Pour lire les données à l'entrée d'un module TDF en provenance d'un module DE, les ports de conversion utilisent deux buffers. Le buffer **in_buffer** dont la taille est déterminée par le calcul de l'ordonnancement statique, et, le cas échéant un buffer **in_delay_buffer** dont la taille est déterminée en fonction des retards attribués à ce port. Contrairement à la PoC SystemC-AMS qui n'utilise qu'une seule fonction pour implémenter la synchronisation, en SystemC MDVP la synchronisation est effectuée par 2 fonctions illustrées à la Figure C.23 :

ⓐ read_sc_signal(t) : qui est appelée par le simulateur pour lire au temps DE t, une valeur issue du canal DE relié au port de conversion p; puis écrire cette valeur dans le buffer **in_buffer** instancié par le port p.

ⓑ read(id) : qui est appelée par le concepteur du modèle pour lire une donnée du port de conversion p. Cette fonction prend comme argument l'index id de l'échantillon à lire. Cet index doit être inférieur au taux de sur-échantillonnage attribué au port d'entrée p.

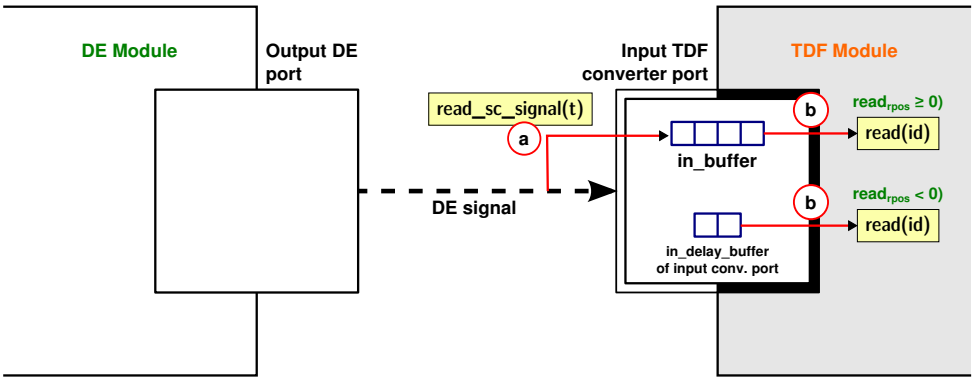

Figure C.23: Spécification des ports de conversion à l'entrée d'un module TDF en SystemC MDVP.

Pour transmettre les données depuis un module TDF vers un module DE, on définit au sein du MoC TDF des ports de conversion de sortie. Ces ports contiennent un buffer **out_buffer**. La Figure C.24, illustre que dans ce cas également, nous avons défini 2 fonctions pour communiquer entre TDF et DE. Il s'agit de :

ⓐ write(val, id): appelée par le concepteur du modèle pour effectuer l'écriture d'une donnée TDF sur le port de conversion p. Cette méthode prend comme argument la valeur val de l'échantillon à enregistrer dans le buffer **out_buffer** du port p, et l'index id de l'échantillon à écrire.

ⓑ write_sc_signal(t): pour lire un échantillon depuis le buffer **out_buffer** instancié dans le port de conversion p; puis écrire à l'instant DE t, une valeur sur le canal DE auquel est relié au port de conversion de sortie p.

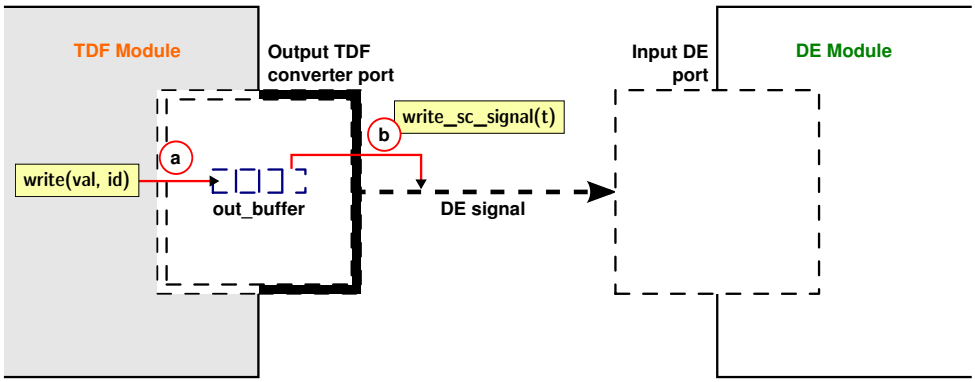

Figure C.24: Spécification des ports de conversion à la sortie d'un module TDF en SystemC MDVP..

C.6.3. L'élaboration et la simulation du MoC TDF de SystemC MDVP

Le *solveur DE-TDF* est, quant à lui, chargé de traiter la synchronisation temporelle entre les MoC DE et TDF. A ce titre, il doit implémenter les fonctions abstraites d'élaboration et de simulation. Ces fonctions sont présentées à la Figure C.25.

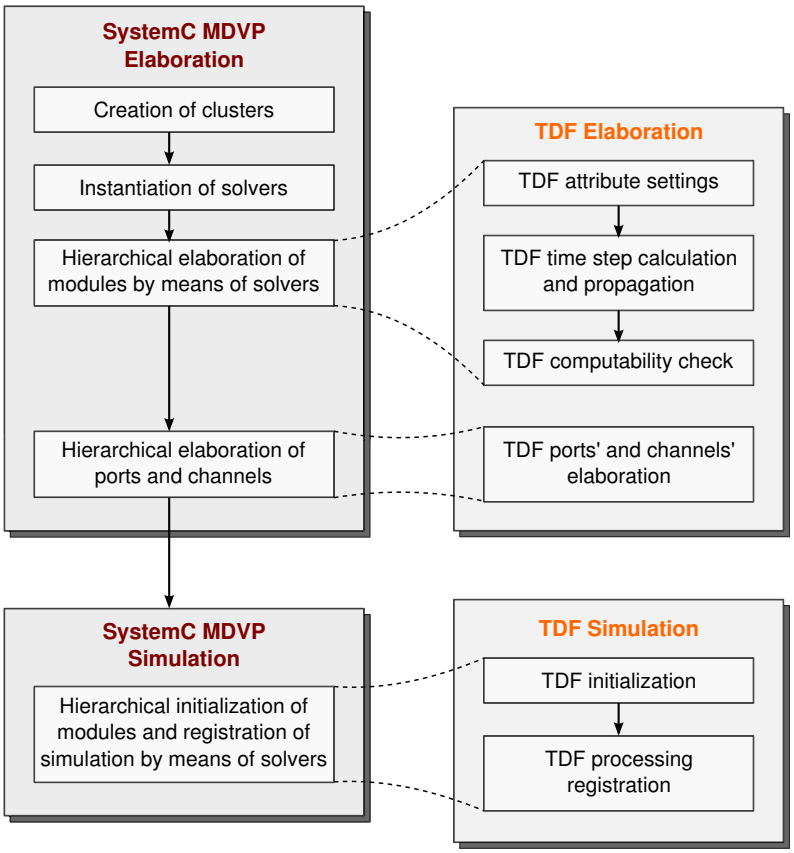

Figure C.25: Phases d'élaboration et de simulation du MoC TDF en interaction avec DE dans SystemC MDVP.

La phase d'élaboration du MoC TDF réalise les opérations suivantes:

- **TDF attribute settings**. C'est une fonction offerte au concepteur de modèle pour imposer les pas de temps au sein des clusters. A la fin de cette étape, le simulateur vérifie qu'il y a au moins un pas de temps défini pour chaque cluster.

- **TDF time step calculation and propagation**. Cette fonction effectue calcul et la propagation des pas de temps TDF à travers les ports et modules de chaque cluster. Cette fonction consiste à :

  b.1 **Propager le pas de temps depuis un module M** vers chaque port m du module, selon l'équation C.3 où Tp_{m} est le pas de temps du port, Tm_{M} est le pas de temps du module et R_{m} est le taux de sur-échantillonnage attribué au port m.

$$Tp_{\mathsf{m}} = \frac{Tm_{\mathsf{M}}}{R_{\mathsf{m}}} \tag{C.3}$$

  b.2 **Propager le pas de temps depuis un port m** à son module et aux autres ports qui lui sont reliés :

  * Au module M, qui contient m, suivant l'équation C.4. Puis recommencer la propagation à partir du module.

$$Tm_{\mathsf{M}} = Tp_{\mathsf{m}} * R_{\mathsf{m}} \tag{C.4}$$

  * A chacun des autres ports n reliés à m, suivant l'équation C.5, tant que le port m n'est pas un port de conversion. Puis recommencer la propagation à partir du port qui vient de recevoir le pas temps.

$$Tp_{\mathsf{n}} = Tp_{\mathsf{m}} \tag{C.5}$$

La Figure C.26 présente la propagation des pas de temps dans le cas de l'exemple du modèle DE-TDF de la Figure C.6(a). Le point initial est indiqué en rouge. Quand, à l'issue de la phase d'attribution des pas de temps (Figure C.25), un cluster se voit attribuer plusieurs pas de temps, le simulateur vérifie que ces temps sont cohérents, c'est à dire qu'ils suivent les équations C.3 à C.5.

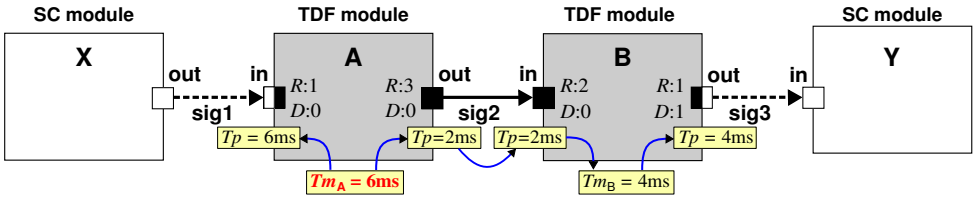

Figure C.26: Exemple du calcul et de la propagation des pas de temps dans un cluster TDF en SystemC MDVP.

- **TDF computability check**. Cette fonction effectue le calcul de l'ordonnancement statique de chaque cluster TDF. Ce calcul est réalisé en 2 étapes. La première repose sur une analyse de type SDF [35], pour vérifier la cohérence des temps et le nombre d'exécution de chaque module au sein d'une période d'un cluster. La deuxième étape construit un réseau de Petri coloré temporisé (CNP) équivalent (Cf. chapitre 4 du document anglais, ou la section C.4.4 de ce chapitre),

pour effectuer l'analyse de synchronisation qui consiste à détecter et corriger les problèmes de synchronisation éventuels entre DE et TDF. Le résultat de cette étape est un ordonnancement valide non seulement des modules TDF au sein de leur cluster, mais aussi des événements induits par l'interaction DE-TDF.

- **TDF ports' and channels' elaboration**. Cette fonction concerne les canaux et les ports de conversion. Elle consiste à déterminer les buffers requis et leur taille, en s'appuyant sur le modèle équivalent CPN construit à l'étape précédente.

Quant à la phase de simulation du MoC TDF, elle réalise les opérations suivantes:

- **TDF initialization**. C'est une fonction offerte au concepteur pour imposer des valeurs initiales à certains échantillons.

- **TDF Processing Registration**. Cette phase est l'exécution des processus des modules TDF. Elle utilise la fonction sc_spawn() du standard SystemC pour créer et enregistrer un processus dynamique [24] après l'appel de sc_start(). En ce qui concerne le MoC TDF, un tel processus est chargé de l'exécution de l'ordonnancement calculé à la fin de la phase d'élaboration, issu de l'analyse du modèle CPN équivalent.

C.6.4. L'implémentation du MoC TDF de SystemC MDVP

L'implémentation du MoC TDF a été réalisée suivant la méthode générique introduite au chapitre 5, Section 5.7. La hiérarchie des classes du MoC TDF est présentée à la Figure C.27. Ces classes héritent des classes du noyau SystemC MDVP.

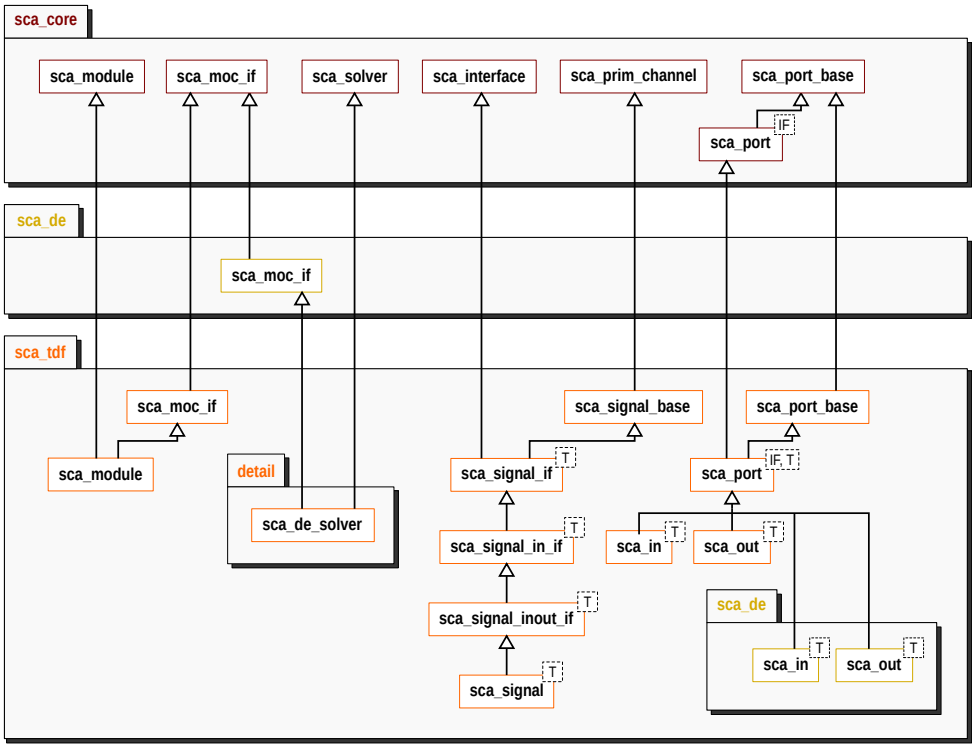

Figure C.27: Vue d'ensemble des classes du MoC TDF de SystemC MDVP.

C.6.5. Exemple

Pour mettre en évidence les avantages du MoC TDF du simulateur SystemC MDVP, nous avons comparé les résultats de simulation de l'exemple Figure C.26 obtenus d'une part avec la PoC SystemC-AMS (Figure C.28) et d'autre part avec SystemC MDVP (Figure C.29).

```
SystemC AMS extensions 2.0 Version: 2.0_beta2 --- BuildRevision: 1808
Copyright (c) 2010-2014  by Fraunhofer-Gesellschaft
Institut Integrated Circuits / EAS
Licensed under the Apache License, Version 2.0

----- Module A / Executing set_attributes()
----- Module B / Executing set_attributes()                    } 1

Info: SystemC-AMS:
      2 SystemC-AMS modules instantiated
      1 SystemC-AMS views created
      2 SystemC-AMS synchronization objects/solvers instantiated

Info: SystemC-AMS:
      1 dataflow clusters instantiated
        cluster 0:
          2 dataflow modules/solver, contains e.g. module: A
          5 elements in schedule list,
          12 ms cluster period,
          ratio to lowest:  3                  e.g. module: B
          ratio to highest: 2 sample time  e.g. module: A
          1 connections to SystemC de, 1 connections from SystemC de

----- Module A / Executing initialize()
----- Module B / Executing initialize()                        } 2

----- Module A / Reading input converter port (sample_id = 0)
----- Module A / Writing output port (sample_id = 0)
----- Module A / Writing output port (sample_id = 1)
----- Module A / Writing output port (sample_id = 2)           } 3

----- Module B / Reading input port (sample_id = 0)
----- Module B / Reading input port (sample_id = 1)
----- Module B / Writing output converter port (sample_id = 0) } 4

----- Module A / Reading input converter port (sample_id = 0)
----- Module A / Writing output port (sample_id = 0)
----- Module A / Writing output port (sample_id = 1)
----- Module A / Writing output port (sample_id = 2)           } 5

----- Module B / Reading input port (sample_id = 0)
----- Module B / Reading input port (sample_id = 1)            } 6

                                                          SC-AMS error
Error: SystemC-AMS: sca-de synchronization failed in: 0
../../../../../src/scams/impl/core/sca_solver_base.cpp line: 526 current sca-time: 4 ms
current sc-time: 6 ms sca-next-time:   8 ms insert da delay of at least: 2 ms in: B.out

In file: ../../../../../src/scams/impl/core/sca_solver_base.cpp:544
In process: sca_implementation_0.cluster_process_0 @ 6 ms
```

Figure C.28: Exécution du modèle TDF présenté à l'exemple Figure C.26, avec SystemC-AMS.

Simulation avec SystemC MDVP :

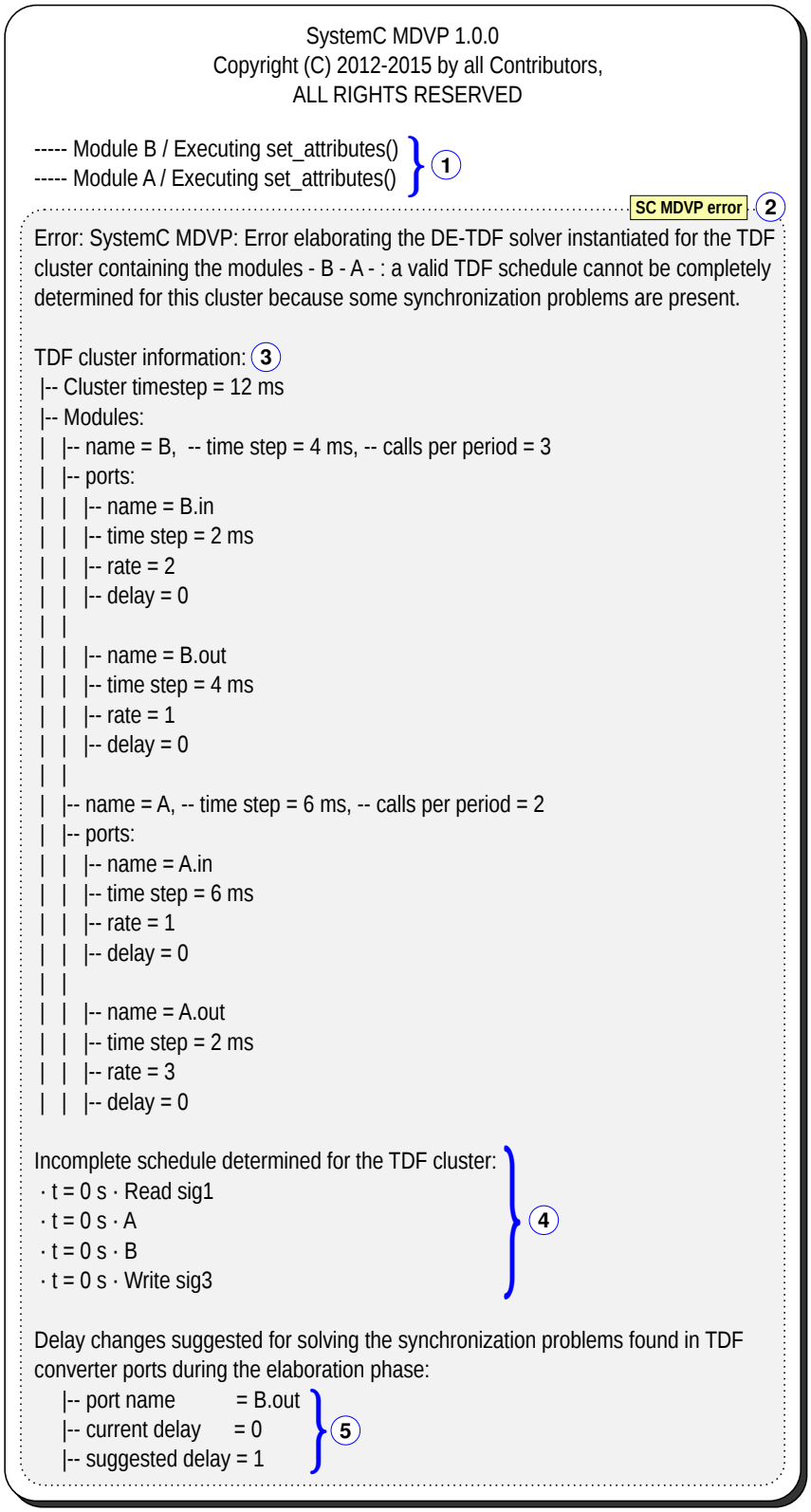

Figure C.29: Exécution du modèle TDF présenté à l'exemple Figure C.26, avec SystemC MDVP.

La trace de simulation obtenue avec SystemC-AMS (Figure C.28) correspond à celle réalisée à la Section C.4.1, lorsque les problèmes de synchronisation sont détectés dans la phase de simulation (Figure C.6). Quant au simulateur SystemC MDVP (Figure C.29), il n'effectue que la phase d'élaboration,

et propose un diagnostic au concepteur de modèle pour corriger les erreurs de modélisation, grâce au support du modèle CPN équivalent (Figure C.11).

C.7. Étude de cas : capteur de vibrations (chapitre 7)

Le chapitre 7 propose d'aborder la modélisation d'un exemple plus complexe, hétérogène, suivant l'approche présentée dans les chapitres précédents. Il s'agit de modéliser un capteur de vibration et son circuit de conditionnement numérique inspiré des études présentées dans [69], [70].

C.7.1. Modélisation du système TDF et équivalent CPN

Le modèle est présenté à la Figure C.30(a). Il comporte 6 modules TDF, dont certains ont des ports à taux d'échantillonnage multiple et un module DE intégré dans une boucle de contre-réaction. Certains modules TDF interagissent donc avec le monde numérique, modélisé dans le domaine DE.

Les modules TDF sont respectivement une source de vibration (**SRC**), le capteur (**SENSOR**), un amplificateur à gain variable (**PGA**), un convertisseur analogique-numérique (**ADC**), un échantillonneur bloqueur (**TDF2DE**), un estimateur d'amplitude (**AAVG**). Le contrôleur de gain (**CTRL**) est modélisé dans le domaine DE comme un automate d'état.

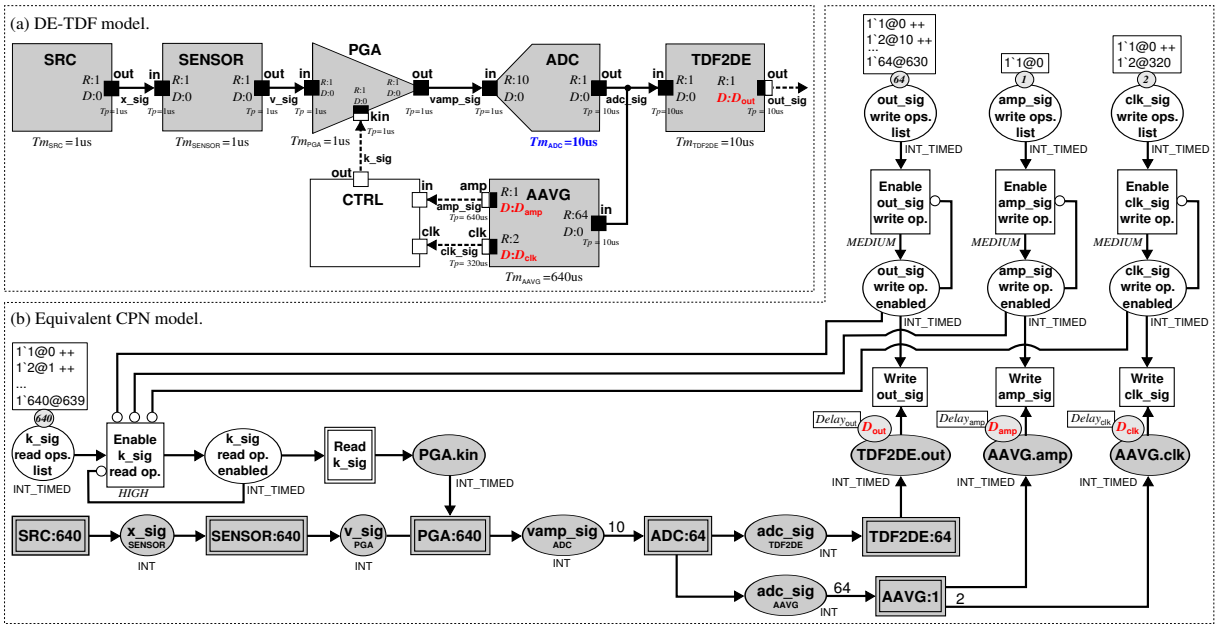

Figure C.30: Modèle SystemC MDVP d'un capteur de vibration et son modèle équivalent CPN.

Comme nous l'avons vu aux chapitres précédents, le simulateur commence par une phase d'élaboration. Il s'agit de procéder à :

- La création des clusters TDF (un seul cluster dans cet exemple) et l'instanciation des solveurs. Ici le solveur est DE-TDF.
- L'élaboration des modules par les solveurs qui consiste à :

- attribuer des pas de temps et des taux de sur-échantillonnage imposés par le concepteur du modèle. Il s'agit par exemple du pas de temps $Tm_{\mathsf{ADC}} = 10\,\mu s$, spécifié dans le module **ADC**; du taux d'échantillonnage $R = 64$, défini sur le port d'entrée TDF **in** du module **AAVG**; ou du taux d'échantillonnage $R = 2$, défini sur le port de conversion de sortie **clk** de ce même module.
- propager les pas de temps aux modules et aux ports non affectés comme cela est indiqué à la Figure C.30(a).
- vérifier dans un premier temps l'existence d'un ordonnancement et établir, le cas échéant, cet ordonnancement (analyse SDF) pour calculer la prériode du cluster TDF. Ici la période du cluster est déterminée par l'Equation C.6.

$$Tcls = Tm_{\mathrm{j}} \cdot q_{\mathrm{j}} \tag{C.6}$$

$$Tcls = Tm_{\mathsf{SRC}} \cdot q_{\mathsf{SRC}} = Tm_{\mathsf{ADC}} \cdot q_{\mathsf{ADC}} = Tm_{\mathsf{AAVG}} \cdot q_{\mathsf{AAVG}}$$
$$Tcls = 1\,\mu s \cdot 640 = 10\,\mu s \cdot 64 = 640\,\mu s \cdot 1$$
$$Tcls = 640\,\mu s$$

  Puis, dans un second temps construire le réseau CPN équivalent (présenté à la Figure C.30(b) et procéder à l'analyse des erreurs de modélisation qui vont entraîner des problèmes de synchronisation.

- L'élaboration des ports et des canaux.

C.7.2. Résultats d'analyse et de simulation par SystemC MDVP

L'analyse des erreurs de modélisation sur le réseau CPN équivalent (Figure C.30(b)) conduisant à des problèmes de synchronisation est illustrée à la Figure C.31.

La trace d'exécution du simulateur fait apparaître que 3 erreurs ont été détectées et que des modifications de retards sur certains ports sont proposées pour corriger ces erreurs.

Lorsque le modèle a été corrigé par son concepteur (pour attribuer les retards $D_{\mathsf{out}} = 1$, $D_{\mathsf{amp}} = 1$ et $D_{\mathsf{clk}} = 2$), et que l'exécution du simulateur SystemC MDVP est relancée, nous obtenons les chonogrammes présentés à la Figure C.32.

- La source produit un signal sinusoïdal ($\mathsf{x}_{\mathsf{sig}}$) d'amplitude 4 µm, d'offset −8 µm et de fréquence pouvant prendre les valeurs 2 kHz, 4 kHz, et 8 kHz.
- Le capteur de vibration génère un signal ($\mathsf{v}_{\mathsf{sig}}$) proportionnel à la vitesse de vibration.
- Ce signal est amplifié ($\mathsf{vamp}_{\mathsf{sig}}$) d'un facteur gain ($2^{\mathsf{ksig}}$), et numérisé ($\mathsf{adc}_{\mathsf{sig}}$). Le seuil de saturation vaut ±5 V.
- Le signal DE ($\mathsf{amp}_{\mathsf{sig}}$) est la moyenne de la valeur absolue de 64 échantillons reçus de l'**ADC**.

L'étude de ce cas a permis de présenter plusieurs propriétés du simulateur SystemC MDVP :

- Les problèmes de synchronisation entre le cluster TDF et le domaine DE ont été détectées et corrigées avant la simulation effective.
- Le concepteur reçoit une notification unique qui récapitule toutes les propositions pour corriger les erreurs de modélisation qui entraînent des erreurs de synchronisation du cluster TDF.
- Les clusters qui présentent des taux d'échantillonnage multiples et des boucles avec le domaine DE, requièrent sur leurs ports des retards non nuls pour établir un ordonnancement valide.

C.8. Conclusions et perspectives (chapitre 8)

Le chapitre 8 du document anglais présente les conclusions de cette thèse et les perpectives.

Cette thèse a abordé la modélisation et la simulation de systèmes hétérogènes, multi-disciplines et multi-domaines temporels dans l'idée de fournir un environnement de simulation SystemC MDVP pour développer des prototypes virtuels.

SystemC MDVP fournit de nouveaux services par rapport à la preuve de concept existante SystemC-AMS :

- **Une analyse et une formalisation des interactions DE-TDF**. Grâce à la modélisation par réseau de Petri coloré temporisé, il est possible de modéliser les interactions DE-TDF et de détecter d'éventuels problèmes de synchronisation avant la phase d'exécution effective de la simulation. Lorsque les modèles sont validés, la simulation s'effectue sans interruption jusqu'à la fin.
- **Une approche générique et systématique pour la synchronisation, l'élaboration et la simulation de MoCs**. SystemC MDVP met en oeuvre une méthode hiérarchique de simulation permettant de synchroniser plusieurs modèles de calculs (MoC) liés par des relations de paires maître-esclave.
- **Ajout d'un MoC**. SystemC MDVP inclut une méthode pour ajouter un nouveau modèle de calcul. Elle nécessite la définition des composants du MoCs : modules, ports et canaux; ainsi que des éléments spécifiques pour traiter la synchronisation : les ports de conversion pour les données et les solveurs pour traiter la synchronisation temporelle. Cette méthode permet de ne pas modifier le comportement des MoCs déjà définis dans le simulateur.

Les perspectives de ce travail sont nombreuses, citons en particulier :

- L'extension du MoC TDF pour réaliser toutes les fonctionnalités définies par le standard SystemC AMS.
- L'ajout de nouveaux MoCs.
- L'ajout de fonctionnalités pour le test et la vérification du comportement de systèmes hétérogènes.

```
SystemC MDVP 1.0.0
Copyright (C) 2012-2015 by all Contributors,
ALL RIGHTS RESERVED
SC MDVP error
Error: SystemC MDVP: Error elaborating the TDF-DE solver instantiated for the TDF
cluster containing the modules - TDF2DE - AAVG - ADC - PGA - SENSOR - SRC - :
a valid TDF schedule cannot be completely determined for this cluster because
some synchronization problems are present.

TDF cluster information:
 |-- Cluster timestep = 640 us
 |-- Modules:
...
 |   |-- name = AAVG
 |   |-- time step = 640 us
 |   |-- calls per period = 1
 |   |-- ports:
 |   |   |-- name = AAVG.in
 |   |   |-- time step = 10 us
 |   |   |-- rate = 64
 |   |   |-- delay = 0
 |   |   |
 |   |   |-- name = AAVG.clk
 |   |   |-- time step = 320 us
 |   |   |-- rate = 2
 |   |   |-- delay = 0
 |   |   |
 |   |   |-- name = AAVG.out
 |   |   |-- time step = 640 us
 |   |   |-- rate = 1
 |   |   |-- delay = 0
...

Incomplete schedule determined for the TDF cluster:
 · t = 0 s · Read k_sig
 · t = 0 s · SRC
 · t = 0 s · SENSOR
 · t = 0 s · SRC
 · t = 0 s · PGA
 · t = 0 s · SENSOR
 · t = 0 s · SRC

Delay changes suggested for solving the synchronization problems found in TDF
converter ports during the elaboration phase:
    |-- port name          = TDF2DE.out
    |-- current delay      = 0
    |-- suggested delay = 1

    |-- port name          = AAVG.clk
    |-- current delay      = 0
    |-- suggested delay = 2

    |-- port name          = AAVG.out
    |-- current delay      = 0
    |-- suggested delay = 1
```

Figure C.31: Execution of the Vibration Sensor Model (with $D_{\mathsf{out}} = 0$, $D_{\mathsf{amp}} = 0$ and $D_{\mathsf{clk}} = 0$) Using SystemC MDVP.

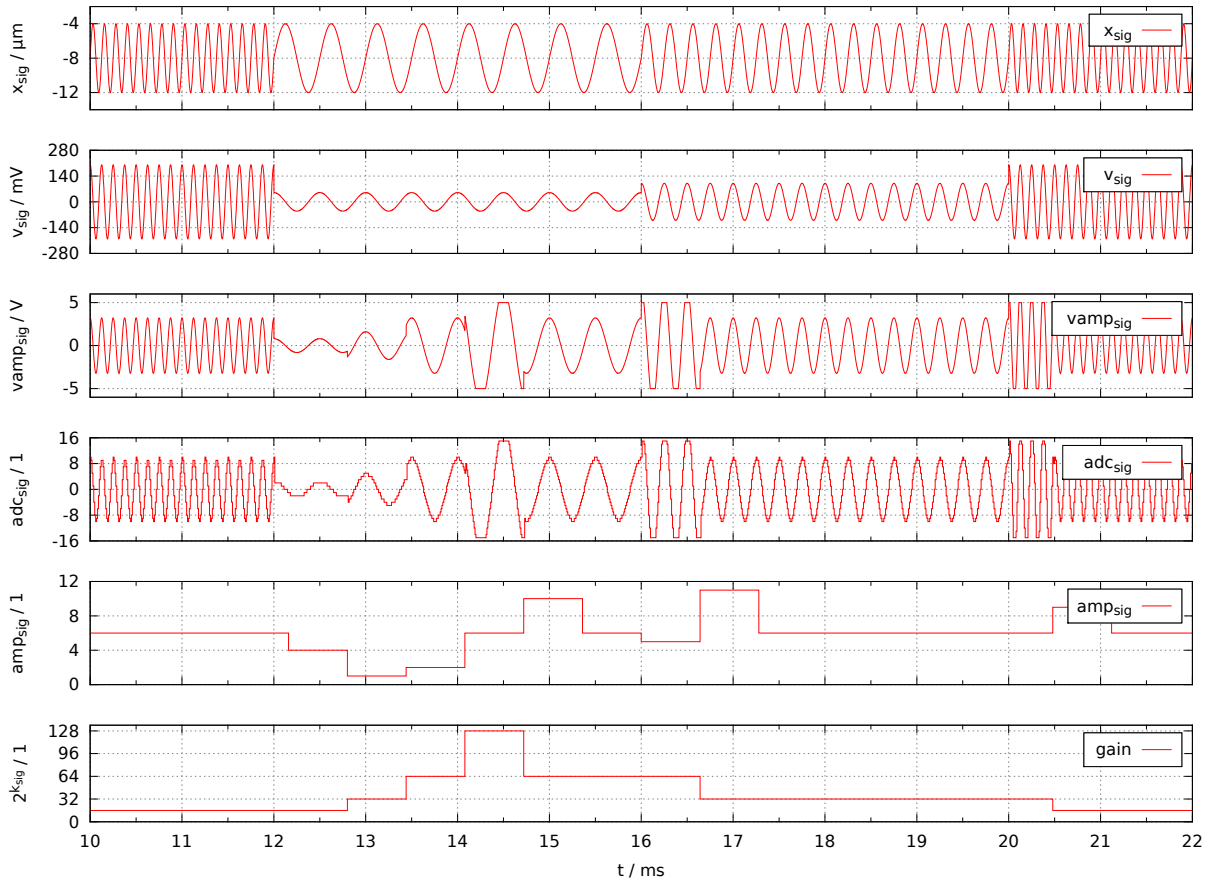

Figure C.32: Traces de simulation SystemC MDVP du modèle du capteur de vibration] avec $D_{\mathsf{out}} = 1$, $D_{\mathsf{amp}} = 1$ et $D_{\mathsf{clk}} = 2$.